*read ref. to see how combo works

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 7/00	A2	(11) International Publication Number: **WO 99/23548** (43) International Publication Date: 14 May 1999 (14.05.99)

(21) International Application Number: PCT/US98/22453

(22) International Filing Date: 22 October 1998 (22.10.98)

(30) Priority Data:

60/063,601	23 October 1997 (23.10.97)	US
60/063,600	23 October 1997 (23.10.97)	US
09/015,084	29 January 1998 (29.01.98)	US
09/049,851	27 March 1998 (27.03.98)	US
09/049,863	27 March 1998 (27.03.98)	US
09/049,893	27 March 1998 (27.03.98)	US
09/049,750	27 March 1998 (27.03.98)	US
09/049,758	27 March 1998 (27.03.98)	US
09/055,916	6 April 1998 (06.04.98)	US
09/098,482	16 June 1998 (16.06.98)	US

(71) Applicant: ADVANCED MICRO DEVICES, INC. [US/US]; One AMD Place, Mail Stop 68, Sunnyvale, CA 94088–3453 (US).

5,7,8, 11-15,18-21, 23-31, 33-35 — Pg51, ¶8 - Pg52 line10
Pg53, line20 - Pg 55 line14
Pg55, line31 - Pg56, line17

(72) Inventors: OBERMAN, Stuart, F.; 265 G. South Bay View Avenue #G, Sunnyvale, CA 94086 (US). JUFFA, Norbert; 460 Navaro Way #201, San Jose, CA 95134 (US). ROBERTS, Mark, E.; 2626 Ohlone Drive, San Jose, CA 95132 (US). WEBER, Frederick, D.; 1137 Settle Avenue, San Jose, CA 95125 (US). CHERUKURI, Ravi, Krishna; 952 Big Bear Court, Milpitas, CA 95035 (US). RAMANI, Krishnan; 655 South Fair Oaks Avenue #G201, Sunnyvale, CA 94086 (US).

(74) Agent: KIVLIN, B., Noël; Conley, Rose & Tayon, P.C., P.O. Box 398, Austin, TX 78767–0398 (US).

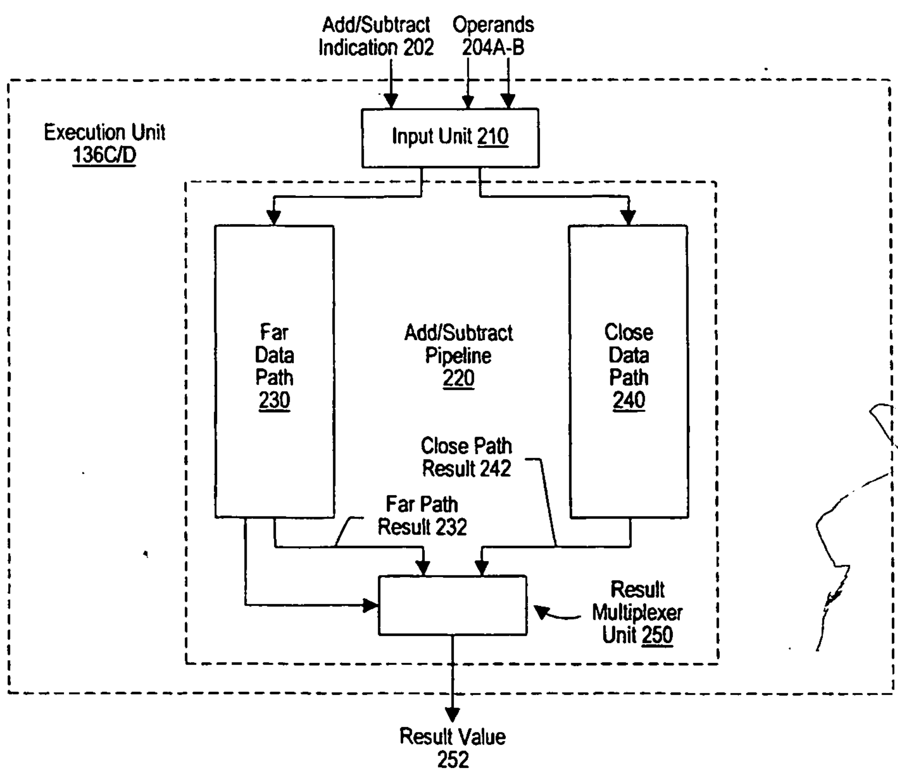
(81) Designated States: CN, JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published

Without international search report and to be republished upon receipt of that report.

(54) Title: MULTIFUNCTION FLOATING POINT ADDITION/SUBTRACTION PIPELINE AND BIPARTITE LOOK–UP TABLE

(57) Abstract



An add/subtract pipeline has far and close data paths. The far data path handles effective addition operations, and effective subtraction operations for operands having an absolute exponent difference greater than one. The close data path handles all other effective subtraction operations. Selection of the output value in the close data path effectuates the round–to–nearest operation. Floating point–to–integer conversion may be executed in the far data path integer–to–floating point instructions in the close data path. The execution unit may include a pluraliy of add/subtract pipelines, allowing vectored add, subtract, and integer/floating point conversion instructions to be performed. Additional arithmetic instructions (such as reverse subtract and accumulate functions minimum/maximum and comparison) may also be implemented. A method for generating entries for a bipartite look–up table having base and difference table portions is also disclosed. So is a multi–function look–up table.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

TITLE: **Multifunction Floating Point Addition/Subtraction Pipeline And Bipartite Look-up Table**

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to floating point arithmetic within microprocessors, and more particularly to an add/subtract pipeline and multifunction bipartite look-up table within a floating point arithmetic unit.

2. Description of the Related Art

Add/Subtract Pipeline

Numbers may be represented within computer systems in a variety of ways. In an integer format, for example, a 32-bit register may store numbers ranging from 0 to 2^{32}-1. (The same register may also signed numbers by giving up one order of magnitude in range). This format is limiting, however, since it is incapable of representing numbers which are not integers (the binary point in integer format may be thought of as being to the right of the least significant bit in the register).

To accommodate non-integer numbers, a *fixed point* representation may be used. In this form of representation, the binary point is considered to be somewhere other than to the right of the least significant bit. For example, a 32-bit register may be used to store values from 0 (inclusive) to 2 (exclusive) by processing register values as though the binary point is located to the right of the most significant register bit. Such a representation allows (in this example) 31 registers bit to represent fractional values. In another embodiment, one bit may be used as a sign bit so that a register can store values between -2 and +2.

Because the binary point is fixed within a register or storage location during fixed point arithmetic operations, numbers with differing orders of magnitude may not be represented with equal precision without scaling. For example, it is not possible to represent both 1101b (13 in decimal) and .1101 (.8125 in decimal) using the same fixed point representation. While fixed point representation schemes are still quite useful, many applications require a larger dynamic range (the ratio of the largest number representation to the smallest, non-zero, number representation in a given format).

In order to solve this problem of dynamic range, floating point representation and arithmetic is widely used. Generally speaking, floating point numeric representations include three parts: a sign bit, an unsigned fractional number, and an exponent value. The most widespread floating point format in use today, IEEE standard 754 (single precision), is depicted in Fig. 1.

Turning now to Fig. 1, floating point format 2 is shown. Format 2 includes a sign bit 4 (denoted as S), an exponent portion 6 (E), and a mantissa portion 8 (F). Floating point values represented in this format have a value V, where V is given by:

$$V = (-1)^S \cdot 2^{E-bias} \cdot (1.F). \quad (1)$$

Sign bit S represents the sign of the entire number, while mantissa portion F is a 23-bit number with an implied leading 1 bit (values with a leading one bit are said to be "normalized"). In other embodiments, the leading one bit may be explicit. Exponent portion E is an 8-bit value which represents the true exponent of the number V offset by a predetermined bias. A bias is used so that both positive and negative true exponents of floating point numbers may be easily compared. The number 127 is used as the bias in IEEE standard 754. Format 2 may thus accommodate numbers having exponents from -127 to +128. Floating point format 2 advantageously allows 24 bits of representation within each of these orders of magnitude.

Floating point addition is an extremely common operation in numerically-intensive applications. (Floating point subtraction is accomplished by inverting one of the inputs and performing addition). Although floating point addition is related to fixed point addition, two differences cause complications. First, an exponent value of the result must be determined from the input operands. Secondly, rounding must be performed. The IEEE standard specifies that the result of an operation should be the same as if the result were computed exactly, and then rounded (to a predetermined number of digits) using the current rounding mode. IEEE standard 754 specifies four rounding modes: round to nearest, round to zero, round to $+\infty$, and round to $-\infty$. The default mode, round to nearest, chooses the even number in the event of a tie.

Turning now to Fig. 2, a prior art floating point addition pipeline 10 is depicted. All steps in pipeline 10 are not performed for all possible additions. (That is, some steps are optional for various cases of inputs). The stages of pipeline 10 are described below with reference to input values A and B. Input value A has a sign bit A_S, an exponent value A_E, and a mantissa value A_F. Input value B, similarly, has a sign bit B_S, exponent value B_E, and mantissa value B_F.

Pipeline 10 first includes a stage 12, in which an exponent difference E_{diff} is calculated between A_E and B_E. In one embodiment, if E_{diff} is calculated to be negative, operands A and B are swapped such that A is now the larger operand. In the embodiment shown in Fig. 2, the operands are swapped such that E_{diff} is always positive.

In stage 14, operands A and B are aligned. This is accomplished by shifting operand B E_{diff} bits to the right. In this manner, the mantissa portions of both operands are scaled to the same order of magnitude. If $A_E=B_E$, no shifting is performed; consequently, no rounding is needed. If $E_{diff}>0$, however, information must be maintained with respect to the bits which are shifted rightward (and are thus no longer representable within the predetermined number of bits). In order to perform IEEE rounding, information is maintained relative to 3 bits: the guard bit (G), the round bit (R),and the sticky bit (S). The guard bit is one bit less significant than the least significant bit (L) of the shifted value, while the round bit is one bit less significant the guard bit. The sticky bit is the logical-OR of all bits less significant than R. For certain cases of addition, only the G and S bits are needed.

In stage 16, the shifted version of operand B is inverted, if needed, to perform subtraction. In some embodiments, the signs of the input operands and the desired operation (either add or subtract) are examined in order to determine whether effective addition or effective subtraction is occurring. In one embodiment, effective addition is given by the equation:

$$EA = A_S \oplus B_S \oplus op, \quad (2)$$

where op is 0 for addition and 1 for subtraction. For example, the operation A minus B, where B is negative, is equivalent to A plus B (ignoring the sign bit of B). Therefore, effective addition is performed. The inversion in stage 16 may be either of the one's complement or two's complement variety.

In stage 18, the addition of operand A and operand B is performed. As described above, operand B may be shifted and may be inverted as needed. Next, in stage 20, the result of stage 18 may be recomplemented, meaning that the value is returned to sign-magnitude form (as opposed to one's or two's complement form).

Subsequently, in stage 22, the result of stage 20 is normalized. This includes left-shifting the result of stage 20 until the most significant bit is a 1. The bits which are shifted in are calculated according to the values of G, R, and S. In stage 24, the normalized value is rounded according to nearest rounding mode. If S includes the R bit OR'ed in, round to nearest (even) is given by the equation:

$$RTN = G(L + S). \quad (3)$$

If the rounding performed in stage 24 produces an overflow, the result is post-normalized (right-shifted) in stage 26.

As can be seen from the description of pipeline 10, floating point addition is quite complicated. This operation is quite time-consuming, also, if performed as shown in Fig. 2: stage 14 (alignment) requires a shift, stage 18 requires a full add, stage 20 (recomplementation) requires a full add, stage 22 requires a shift, and stage 24 (rounding) requires a full add. Consequently, performing floating point addition using pipeline 10 would cause add/subtract operations to have a similar latency to floating point multiplication. Because of the frequency of floating point addition, higher performance is typically desired. Accordingly, most actual floating point add pipeline include optimizations to pipeline 10.

Turning now to Fig. 3, a prior art floating point pipeline 30 is depicted which is optimized with respect to pipeline 10. Broadly speaking, pipeline 30 includes two paths which operate concurrently, far path 31A and close path 31B. Far path 31A is configured to perform all effective additions. Far path 31A is additionally configured to perform effective subtractions for which $E_{diff} > 1$. Close path 31B, conversely is configured to perform effective subtractions for which $E_{diff} \leq 1$. As with Fig. 2, the operation of pipeline 30 is described with respect to input values A and B.

Pipeline 30 first includes stage 32, in which operands A and B are received. The operands are conveyed to both far path 31A and close path 31B. Results are then computed for both paths, with the final result selected in accordance with the actual exponent difference. The operation of far path 31A is described first.

In stage 34 of far path 31A, exponent difference E_{diff} is computed for operands A and B. In one embodiment, the operands are swapped if $A_E > B_E$. If E_{diff} is computed to be 0 or 1, execution in far path 31A is cancelled, since this case is handled by close path 31B as will be described below. Next, in stage 36, the input values are aligned by right shifting operand B as needed. In stage 38, operand B is conditionally inverted in the case of effective subtraction (operand B is not inverted in the case of effective addition). Subsequently, in stage 40, the actual addition is performed. Because of the restrictions placed on far path ($E_{diff} > 1$), the result of stage

40 is always positive. Thus, no recomplementation step is needed. The result of stage 40 is instead rounded and post-normalized in stages 42 and 44, respectively. The result of far path 31A is then conveyed to stage 58.

In stage 46 of close path 31B, exponent difference E_{diff} is calculated in stage 46. If E_{diff} is computed to less than equal to 1, execution continues in close path 31B with stage 48. In one embodiment, operands A and B are swapped (as in one embodiment of far path 31A) so that $A_E \geq B_E$. In stage 48, operand B is inverted to set up the subtraction which is performed in stage 50. In one embodiment, the smaller operand is also shifted by at most one bit. Since the possible shift amount is low, however, this operation may be accomplished with greatly reduced hardware.

The output of stage 50 is then recomplemented if needed in stage 52, and then normalized in stage 54. This result is rounded in stage 56, with the rounded result conveyed to stage 58. In stage 58, either the far path or close path result is selected according to the value of E_{diff}.

It is noted that in close path 31B, stage 52 (recomplementation) and stage 56 (rounding) are mutually exclusive. A negative result may only be obtained in close path 31B in the case where $A_E=B_E$ and $A_P<B_P$. In such a case, however, no bits of precision are lost, and hence no rounding is performed. Conversely, when shifting occurs (giving rise to the possibility of rounding), the result of stage 50 is always positive, eliminating the need for recomplementation in stage 52.

The configuration of pipeline 30 allows each path 31 to exclude unneeded hardware. For example, far path 31A does not require an additional adder for recomplementation as described above. Close path 31B eliminates the need for a full shift operation before stage 50, and also reduces the number of add operations required (due to the exclusivity of rounding and recomplementation described above).

Pipeline 30 offers improved performance over pipeline 10. Because of the frequency of floating point add/subtract operations, however, a floating point addition pipeline is desired which exhibits improved performance over pipeline 30. Improved performance is particularly desired with respect to close path 31B.

Multifunction Bipartite Look-Up Table

Floating-point instructions are used within microprocessors to perform high-precision mathematical operations for a variety of numerically-intensive applications. Floating-point arithmetic is particularly important within applications that perform the rendering of three-dimensional graphical images. Accordingly, as graphics processing techniques grow more sophisticated, a corresponding increase in floating-point performance is required.

Graphics processing operations within computer systems are typically performed in a series of steps referred to collectively as the graphics pipeline. Broadly speaking, the graphics pipeline may be considered as having a front end and a back end. The front end of receives a set of vertices and associated parameters which define a graphical object in model space coordinates. Through a number of steps in the front end of the pipeline, these vertices are assembled into graphical primitives (such as triangles) which are converted into screen space coordinates. One distinguishing feature of these front-end operations (which include view transformation, clipping, and perspective division) is that they are primarily performed using floating-point

numbers. The back end of the pipeline, on the other hand, is typically integer-intensive and involves the rasterization (drawing on a display device) of geometric primitives produced by the front end of the pipeline.

High-end graphics systems typically include graphics accelerators coupled to the microprocessor via the system bus. These graphics accelerators include dedicated hardware specifically designed for efficiently performing operations of the graphics pipeline. Most consumer-level graphics cards, however, only accelerate the rasterization stages of the graphics pipeline. In these systems, the microprocessor is responsible for performing the floating-point calculations in the initial stages of the graphics pipeline. The microprocessor then conveys the graphics primitives produced from these calculations to the graphics card for rasterizing. For such systems, it is clear that increased microprocessor floating-point performance may result in increased graphics processing capability.

One manner in which floating-point performance may be increased is by optimizing the divide operation. Although studies have shown that division represents less than 1% of all instructions in typical floating-point code sequences (such as SPECfp benchmarks), these instructions occupy a relatively large portion of execution time. (For more information on the division operation within floating-point code sequences, please refer to "Design Issues in Division and Other Floating-Point Operations", by Stuart F. Oberman and Michael J. Flynn, published in *IEEE Transactions on Computers*, Vol. 46, No. 2, February 1997, pp. 154-161). With regard to the front-end stages of the graphics pipeline, division (or, equivalently, the reciprocal operation) is particularly critical during the perspective correction operation. A low-latency divide operation may thus prevent a potential bottleneck and result in increased graphics processing performance.

Additional floating-point performance may be gained by optimization of the reciprocal square root operation (1/sqrt(x)). Most square roots in graphics processing occur in the denominators of fractions, so it is accordingly advantageous to provide a function which directly computes the reciprocal of the square root. Since the reciprocal square root operation is performed during the common procedures of vector normalization and viewing transformations, optimization of this function represents a significant potential performance enhancement.

One means of increasing performance of the reciprocal and reciprocal square root operations is through the use of dedicated floating-point hardware. Because floating-point hardware is relatively large as compared to comparable fixed-point hardware, however, such an implementation may use a significant portion of the hardware real estate allocated to the floating-point unit. An alternate approach is to utilize an existing floating-point element (such as a multiplier) to implement division based on iterative techniques like the Goldschmidt or Newton-Raphson algorithms.

Iterative algorithms for division require a starting approximation for the reciprocal of the divisor. A predetermined equation is then evaluated using this starting approximation. The result of this evaluation is then used for a subsequent evaluation of the predetermined equation. This process is repeated until a result of the desired accuracy is reached. In order to achieve a low-latency divide operation, the number of iterations needed to achieve the final result must be small. One means to decrease the number of iterations in the division operation is to increase the accuracy of the starting approximation. The more accurately the first approximation is determined, then, the more quickly the division may be performed.

Starting approximations for floating-point operations such as the reciprocal function are typically obtained through the use of a look-up table. A look-up table is a read-only memory (ROM) which stores a predetermined output value for each of a number of regions within a given input range. For floating-point functions such as the division operation, the look-up table is located within the microprocessor's floating-point unit. An input range for a floating-point function is typically bounded by a single binade of floating point values (a "binade" refers to a range of numbers between consecutive powers of 2). Input ranges for other floating-point functions, however, may span more than one binade.

Because a single output value is assigned for each region within a function's input range, some amount of error is inherently introduced into the result provided by the table look-up operation. One means of reducing this error is to increase the number of entries in the look-up table. This limits the error in any given entry by decreasing the range of input arguments. Often times, however, the number of entries required to achieve a satisfactory degree of accuracy in this manner is prohibitively large. Large tables have the unfortunate properties of occupying too much space and slowing down the table look-up (large tables take longer to index into than relatively smaller tables).

In order to decrease table size while still maintaining accuracy, "bipartite" look-up tables are utilized. Bipartite look-up tables actually include two separate tables: a base value table and a difference value table. The base table includes function output values (or "nodes") for various regions of the input range. The values in the difference table are then used to calculate function output values located between nodes in the base table. This calculation may be performed by linear interpolation or various other techniques. Depending on the slope of the function for which the bipartite look-up table is being constructed, table storage requirements may be dramatically reduced while maintaining a high level of accuracy. If the function changes slowly, for example, the number of bits required for difference table entries is much less than the number of bits in the base table entries. This allows the bipartite table to be implemented with fewer bits than a comparable naïve table (one which does not employ interpolation).

Prior art bipartite look-up tables provide output values having a minimal amount of maximum relative error over a given input interval. This use of relative error to measure the accuracy of the look-up table output values is questionable, however, because of a problem known as "wobbling precision". Wobbling precision refers to the fact that a difference in the least significant bit of an input value to the look-up table has twice the relative error at the end of a binade than it has at the start of the binade. A look-up table constructed in this manner is thus not as accurate as possible.

It would therefore be desirable to have a bipartite look-up table having output values with improved accuracy.

As described above, increasing the efficiency of the reciprocal and reciprocal square root functions may lead to increased floating-point performance (and thus, increased graphics processing performance). While prior art systems have implemented a single function (such as the reciprocal function) using a look-up table, this does not take advantage of the potential savings of optimizing both the reciprocal and reciprocal square root functions using look-up tables. This potential performance gain is outweighed by additional overhead required by the separate look-up table.

It would therefore be desirable to have a multi-function look-up table which implements both the reciprocal and reciprocal square root functions with minimal overhead. It would further be desirable for the multi-function look-up table to be a bipartite look-up table.

SUMMARY OF THE INVENTION

Add/Subtract Pipeline

The problems outlined above are in large part solved by an execution unit in accordance with the present invention. In one embodiment, an execution unit is provided which is usable to perform effective addition or subtraction upon a given pair of floating point input values. The execution unit includes an add/subtract pipeline having a far data path and a close data path each coupled to receive the given pair of floating point input values. The far data path is configured to perform effective addition as well as effective subtraction upon operands having an absolute exponent difference greater than one. The close data path, on the other hand, is configured to perform effective subtraction upon operands having an absolute exponent difference less than or equal to one. The add/subtract pipeline further includes a result multiplexer unit coupled to receive a result from both the far data path and the close data path. A final output of the result multiplexer unit is selected from the far path result and the close path result according to the actual calculated absolute exponent difference value.

In one embodiment, the far data path includes a pair of right shift units coupled to receive mantissa portions of each of the given pair of floating point input values. The right shift units each receive a shift amount from a corresponding exponent difference unit. The first right shift unit conveys a shift amount equal to the second exponent value minus the first exponent value, while the second right shift unit conveys a shift amount equal to the first exponent value minus the second exponent value. The outputs of the right shift units are then conveyed to a multiplexer-inverter unit, which also receives unshifted versions of the mantissa portions of each of the given pair of floating point input values. The multiplexer-inverter unit is configured to select one of the unshifted mantissa portions and one of the shifted mantissa portions to be conveyed as inputs to an adder unit. The adder inputs conveyed by the multiplexer-inverter unit are aligned in order to facilitate the addition operation. The multiplexer-inverter unit is further configured to invert the second adder input if the effective operation to be performed is subtraction.

The adder unit is configured to add the first and second adder inputs, thereby generating first and second adder outputs. The first adder output is equal to the sum of the two inputs, while the second adder output is equal to the first adder output plus one. One of the two adder outputs is selected according to a far path selection signal generated by a far path selection unit. The far path selection unit is configured to generate a plurality of preliminary far path selection signals. Each of these preliminary far path selection signals corresponds to a different possible normalization of the first adder output. For example, one of the preliminary far path selection signals corresponds to a prediction that the first adder output is properly normalized. Another preliminary far path selection signal corresponds to a prediction that the first adder output is not normalized, while still another select signal indicates that said first adder output has an overflow bit set. One of these

preliminary far path selection signals is selected to be conveyed as the final far path selection signal based on which of these predictions actually occurs.

The far data path further includes a multiplexer-shift unit configured to receive the first and second adder outputs as well as the final far path selection signal. The appropriate adder output is selected, and a one-bit left or right shift may also be performed to properly normalize the result. In the case of a left shift, a guard bit previously shifted out of one of the mantissa values by a right shift unit may be shifted back into the final result. The selected value is conveyed as a mantissa portion of the far data path result value. The exponent portion of the far path result is calculated by a exponent adjustment unit. The exponent adjustment unit is configured to receive the original larger exponent value along with the amount of shifting required for proper normalization (which may be no shift, a one-bit left shift, or a one-bit right shift).

In contrast to a generic floating point addition/subtraction pipeline, the far data path is optimized to perform effective additions. The far data path is additionally optimized to perform effective subtractions on operands having an absolute exponent difference greater than one. This configuration allows the recomplementation step to be avoided, since all operations produce positive results. Furthermore, since adder outputs require at most a one-bit shift, only one full-size shifter is needed in the far data path. This results in improved floating point addition and subtraction performance for the far data path.

In one embodiment, the close data path is coupled to receive mantissa portions of the given pair of floating point input values, as well as two least significant bits of each of the exponent values. The mantissa values are conveyed to a shift-swap unit, which also receives an exponent difference prediction from an exponent prediction unit. The exponent difference prediction is indicative of whether the absolute exponent difference is 0 or 1. It is used to align and swap (if needed) the input mantissa values for conveyance to a close path adder unit. The mantissa values are swapped such that the exponent value associated with the first adder input is greater than or equal to the exponent value associated with the second adder input. The first adder input is not guaranteed to be greater than the second adder input if the exponent values are equal, however. The shift-swap unit is also configured to invert the second adder input since the adder unit within the close data path performs subtraction.

It is further noted that the exponent difference value generated by the exponent prediction unit may be incorrect. This is true since the exponent prediction is based only on a subset of the total number of bits. The result produced by the close data path is thus speculative. The actual exponent difference calculated in the far data path is used to determine whether the result produced by the close data path is valid.

The adder unit within the close data path produces a first and second output value. The first output value is equal to the first adder input plus the second adder input, which is effectively equivalent to the first mantissa portion minus the second mantissa portion. The second output value, on the other hand, is equal to the first output value plus one. Both values are conveyed to a multiplexer-inverter unit. A close path selection signal provided by a close path selection unit is usable to select either the first adder output or the second adder output as a preliminary close path result.

The selection unit includes a plurality of logic sub-blocks, each of which is configured to generate a preliminary close path selection signal indicative of either the first adder output value or the second adder output value. Each of the preliminary close path selection signals corresponds to a different predictions

scenario. For example, a first logic sub-block generates a preliminary close path select signal for the case in which the exponent values are equal and the first mantissa value is greater than the second mantissa value. A second logic sub-block generates a select signal for the case in which the exponent values are equal and the first mantissa value is less than the second mantissa value. A third logic sub-block corresponds to the case in which the first exponent value is greater than the second exponent value and the first adder output is not normalized. The last sub-block corresponds to the case in which the first exponent value is greater than the second exponent value and the first adder output is normalized. Each of the preliminary selection signals is conveyed to a close path selection multiplexer, the output of which is used to select either the first or second adder output as the preliminary close path subtraction result.

The output for the close path selection multiplexer is determined by which of the various predicted cases actually occurs. Accordingly, the close path selection multiplexer receives as control signals the exponent prediction value (indicating whether the exponents are equal or not), the sign value of the first adder output (indicating whether a negative result is present), and the MSB of the first adder output (indicating whether the result is properly normalized or not). The sign value and the MSB value are generated concurrently within both the adder unit and the selection unit. This is accomplished using a carry chain driven by C_{MSB}, the carry in signal to the most significant bit position of the adder unit. This concurrent generation allows faster selection of either the first or second adder outputs. The selection of one of these values effectuates rounding the close path result to the nearest number (an even number is chosen in the event of a tie). This configuration advantageously eliminates the need for a separate adder unit to perform rounding.

If the first adder output is negative, the multiplexer-inverter unit inverts the first adder output to produce the correct result. This occurs for the case in which the exponents are equal and the second mantissa value is greater than the first mantissa value. In any event, the selected close path preliminary subtraction result is then conveyed to a left shift unit for normalization.

The close path preliminary subtraction result conveyed to the left shift unit is shifted according to a predicted shift amount generated by a shift prediction unit. The shift prediction unit includes three leading 0/1 detection units. The first unit, a leading 1 detection unit, generates a first prediction string for the case in which the first exponent value is greater than the second exponent value. The second unit, which performs both leading 0 and 1 detection, generates a second prediction string for the case in which the first and second exponent values are equal. Leading 0 and 1 detection is performed because the result may be positive (leading 1) or negative (leading 0). Finally, the third unit, a leading 1 detection unit, generates a third prediction string for the case in which the second exponent value is greater than the first exponent value. The most significant asserted bits within each of the strings indicates the position of a leading 0 or 1 value.

Each of the three prediction strings are generated concurrently and conveyed to a shift prediction multiplexer. The exponent prediction value generated by the exponent prediction unit within the close data path selects which of the prediction strings is conveyed by the shift prediction multiplexer to a priority encoder. The priority encoder then converts the selected prediction string to a shift amount which is conveyed to the left shift unit within the close data path. The predicted shift amount may in some instances be incorrect by one bit position. For such cases, the close path result is left shifted one place during final selection. The calculated

results of both the far data path and close data path are conveyed to a final result multiplexer, which selects the correct result based upon the calculated actual exponent difference value.

Within the shift prediction unit, the second leading 0/1 detection unit may not be optimized further, since no assumptions may be made regarding its inputs. The first and third prediction units, however, may be optimized, since it is known that the second mantissa to each unit is inverted and shifted one bit rightward with respect to the first mantissa. This means that the results predicted by the first and third detection units are both positive. Hence, only leading 1 detection is desired. Further optimizations may also be made since it is known that subtraction is being performed.

Prediction strings may be formed by assigning a value to each output bit based on the corresponding inputs for that bit position. In standard T-G-Z notation, a T output value represents input values 10 or 01, a G output value represents input values 11, and a Z output value represents output values 00. A leading 1 may thus be detected whenever the pattern T^*GZ^* stops matching in the generated prediction string.

The two leading 1 detection units within the shift prediction unit of the close data path may optimized over prior art designs by recognizing that the MSB of both input operands is 1. (The MSB of the first operand is a 1 since it is normalized, and the MSB of the second operand is also a 1 since the second adder operand is right shifted one place then inverted). This corresponds to an output value of G in the MSB of the prediction string. With a G in the initial position of the prediction string, it may be recognized that the string stops matching whenever Z' (the complement of Z) is found. This condition is realized whenever at least one of the inputs in a given bit position is set.

The optimized leading 1 detection unit includes a pair of input registers and an output register for storing the generated prediction string. The first input register is coupled to receive the first (greater) mantissa value, while the second input register is coupled to receive an inverted version of the second (lesser) mantissa value. The leading 1 detection unit further includes a plurality of logic gates coupled to receive bits from each of the input registers. Each logic gate generates a bit for the final prediction string based on whether one of the inputs is set. The most significant asserted bit in the output prediction string indicates the position of the leading 1 bit.

The add/subtract pipeline may also be configured to perform floating point-to-integer and integer-to-floating point conversions. In one embodiment, the far data path may be used to perform floating point-to-integer conversions, while the close data path performs integer-to-floating point conversions. Both data paths are configured to be as wide as the width of the larger format.

In order to perform floating point-to-integer conversions within the far data path, a shift amount is generated from the maximum integer exponent value and the exponent value of the floating point number to be converted. The floating point mantissa to be converted is then right shifted by the calculated shift amount and conveyed to the multiplexer-inverter unit. The multiplexer-inverter unit conveys the converted mantissa value to the adder unit as the second adder input. The first adder input is set to zero.

As with standard far path operation, the adder unit produces two output values, sum and sum+1. These values are conveyed to the multiplexer-shift unit, where the first adder output (sum) is selected by the far path selection signal. The far path selection unit is configured to select the sum output of the adder unit in response to receiving an indication that a floating point-to-integer conversion is being performed.

The floating point number being converted may greater than the maximum representable integer (or less than the minimum representable integer). Accordingly, comparisons are performed to determine whether overflow or underflow has occurred. If either condition is present, the integer result is clamped at the maximum or minimum value.

In order to perform integer-to-floating point conversions within the close data path, a zero value is utilized as the first operand, while the second operand is the integer value to be converted. The second operand is inverted (since close path performs subtraction) and conveyed along with the zero value to the adder unit. The adder unit, as in standard close path operations, produces two outputs, sum and sum+1.

If the input integer value is positive, the output of the adder unit is negative. Accordingly, the sum output is chosen by the selection unit as the preliminary close path result. This output is then inverted in the multiplexer-inverter unit to produce the correct result. If, on the other hand, the input integer value is negative, the output of the adder unit is positive. The sum+1 output is thus chosen as the preliminary close path result, and the sign of the resulting floating point number is denoted as being negative.

The preliminary close path result is then conveyed to the left shift unit for normalization, which is performed in accordance with a predicted shift amount conveyed from the shift prediction unit. For integer-to-floating point conversion, the prediction string of the second prediction unit (equal exponents) is used. The zero operand and an inverted version of the integer value are conveyed as inputs to the second prediction unit.

The shift amount generated by the shift prediction unit is usable to left align the preliminary close path result (with a possible one-bit correction needed). With alignment performed, the number bits in the floating point mantissa may thus be routed from the output of the left shift unit to form the mantissa portion of the close path result. The exponent portion of the close path result is generated by an exponent adjustment unit.

The exponent adjustment unit is configured to subtract the predicted shift amount from the maximum exponent possible in the integer format. The result (which may also be off by 1) becomes the exponent portion of the close path result. If the dynamic range of the floating point format is greater than the maximum representable integer value, overflows do not occur.

The execution unit may also be configured to include a plurality of add/subtract pipelines each having a far and close data path. In this manner, vectored instructions may be performed which execute the same operations on multiple sets of operands. This is particularly useful for applications such as graphics in which similar operations are performed repeatedly on large sets of data.

In addition to performing vectored add and subtract operations, the execution unit may also be configured to perform vectored floating point-to-integer and integer-to-floating point instructions as described above. The execution unit may still further be configured to perform additional vectored arithmetic operations such as reverse subtract and accumulate functions by appropriate multiplexing of input values to the far and close data paths. Other vectored operations such as extreme value functions and comparison operations may be implemented through appropriate multiplexing of output values.

<u>Multifunction Bipartite Look-Up Table</u>

The related problems concerning lookup tables outlined above are in large part solved by a method for generating entries for a bipartite look-up table which includes a base table portion and a difference table portion.

In one embodiment, these entries are usable to form output values for a given mathematical function (denoted as f(x)) in response to receiving corresponding input values (x) within a predetermined input range. For example, the bipartite look-up table may be used to implement the reciprocal function or the reciprocal square root function, both of which are useful for performing 3-D graphics operations.

The method first comprises partitioning the input range of the function into intervals, subintervals, and sub-subintervals. This first includes dividing the predetermined input range into a predetermined number (I) of intervals. Next, the I intervals are each divided into J subintervals, resulting in I*J subintervals for the input range. Finally, each of the I*J subintervals is divided into K sub-subintervals, for a total of I*J*K sub-subintervals over the input range.

The method next includes generating K difference table entries for each interval in the predetermined input range. Each of the K difference table entries for a given interval corresponds to a particular group of sub-subintervals within the given interval. In one embodiment, this particular group of sub-subintervals includes one sub-subinterval per subinterval of the given interval. Additionally, each sub-subinterval in the particular group has the same relative position within its respective subinterval. For example, one of the K difference table entries for the given interval may correspond to a first group of sub-subintervals wherein each sub-subinterval is the last sub-subinterval within its respective subinterval.

In order to generate a first difference table entry for a selected interval (M), a group of sub-subintervals (N) within interval M is selected to correspond to the first entry. The calculation of the first entry then begins with a current subinterval (P), which is bounded by input values A and B. A midpoint X1 is calculated for current subinterval P such that f(A)-f(X1)=f(X1)-f(B). (By calculating the midpoint in this way, maximum possible absolute error is minimized for all input values within the sub-subinterval). Next, a midpoint X2 is computed in a similar fashion for a predetermined reference sub-subinterval within current subinterval P. (The reference sub-subinterval refers to the sub-subinterval within each subinterval that corresponds to the base table entry). A difference value, f(X1)-f(X2), is then computed for current subinterval P.

In this manner, a difference value is computed for each sub-subinterval in group N. A running total is maintained of each of these difference values. The final total is then divided by J, the number of subintervals in the selected intervals, in order to generate the difference value average for interval M, sub-subinterval group N. In one embodiment, the difference value average is converted into an integer value before being stored to the difference table portion of the bipartite look-up table.

The above-described steps are usable to calculate a single difference table entry for interval M. In order to calculate the remaining difference table entries for the selected interval, each remaining group of sub-subintervals is selected in turn, and a corresponding difference value average is computed. In this manner, the additional K-1 difference table entries may be generated for interval M. Difference table entries for any additional intervals in the predetermined input range are calculated in a similar manner.

The method next includes generating J base table entries for each interval in the predetermined input range. Each of the J base table entries for a given interval corresponds to a particular subinterval within the given interval. For example, one of the J base table entries for the given interval may correspond to the first subinterval of the given interval.

In a similar manner to the difference table computations, a particular interval (M) of the predetermined input range is selected for which to compute the J base table entries. Next, a subinterval of interval M is chosen as a currently selected subinterval P. Typically, the first subinterval is initially chosen as subinterval P.

The method then includes calculating an initial base value, B, where B=f(X2). (As stated above, X2 is the midpoint of the reference sub-subinterval of subinterval P of interval M). Subsequently, a difference value, D, is computed, where D=f(X3). (X3 is the midpoint of the sub-subinterval within subinterval P which is furthest from the reference sub-subinterval. For example, if the reference sub-subinterval is the last sub-subinterval in subinterval P, X3 is computed for the first sub-subinterval in P).

The actual maximum midpoint difference for subinterval P is given by D-B. A reference is then made to the previously computed difference table entry for the sub-subinterval (or, more appropriately, the sub-subinterval group) within interval M which corresponds to the sub-subinterval for which D is computed. Since this value is computed by difference averaging as described above, the difference average differs from the quantity D-B.

The difference of the actual difference value and the average difference value is the maximum error for subinterval P. An adjust value is then computed as a fraction of this maximum error value. (In one embodiment, the adjust value is half of the maximum error value in order to evenly distribute the error over the entire subinterval). The final base value is calculated by adding the adjust value (which may be positive or negative) to the initial base value B. In one embodiment, this final base value may be converted to an integer for storage to the base table portion of the bipartite look-up table. The steps described above are repeated for the remaining subintervals in the selected interval, as well as for the subintervals of the remaining intervals of the predetermined input range.

In one embodiment, the output values of the bipartite look-up table are simply the sum of selected base and difference table entries. If these entries are calculated as described above, the resultant output values of the table will have a minimal amount of possible absolute error. Additionally, this minimized absolute error is achieved within a bipartite table configuration, which allows reduced storage requirements as compared to a naive table of similar accuracy. Furthermore, in an embodiment in which the base and difference values are added to generate the table outputs, this allows the interpolation to be implemented with only the cost of a simple addition. This increases the speed of the table look-up operation, in contrast to prior art systems which often require lengthy multiply-add or multiply instructions as part of the interpolation process.

In one embodiment, a multi-function look-up table is provided for determining output values for a first mathematical function and a second mathematical function. These output values correspond to input values which fall within predetermined input ranges which are divided into intervals. The intervals are in turn further divided into subintervals, with each of the resulting subintervals subdivided into sub-subintervals. . In one embodiment, generated output values have minimized possible absolute error.

In one embodiment, the multi-function look-up table is a bipartite look-up table including a first plurality of storage locations and a second plurality of storage locations. These first plurality of storage locations store base values for the first and second mathematical functions, respectively. Each base value is an output value (for either the first or second function) corresponding to an input region which includes the look-up table input value. In one embodiment, each base value in the first plurality of storage locations corresponds

to one of the subintervals in the predetermined input ranges. The second plurality of storage locations, on the other hand, store difference values for both the first and second mathematical functions. These difference values are used for linear interpolation in conjunction with a corresponding base value. In one embodiment, each of the difference values corresponds to one of a group of sub-subintervals in the predetermined ranges. The selected group of sub-subintervals includes one particular sub-subinterval which includes the look-up table input value.

The multi-function look-up table further includes an address control unit coupled to receive a first set of input signals. This first set of input signals includes a first input value and a signal which indicates whether an output value is to be generated for the first or second mathematical function. The address control unit is configured to generate a first address value from the first set of input signals. This first address value is in turn conveyed to the first plurality of storage locations and the second plurality of storage locations.

In response to receiving the first address value, the first plurality of storage locations is configured to output a first base value. Likewise, the second plurality of storage locations is configured to output a first difference value in response to receiving the first address value. The multi-function look-up table finally includes an output unit coupled to receive the first base value from the first plurality of storage locations and the first difference value from the second plurality of storage locations. The output unit is additionally configured to generate the first output value from the first base value and the first difference value. In one embodiment, the output unit generates the first output value by adding the first difference value to the first base value.

By employing a multi-function look-up table, a microprocessor may enhance the performance of both the reciprocal and reciprocal square root functions. Floating-point and graphics processing performance is thus advantageously enhanced.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

Fig. 1 depicts the format of a single precision floating point number according to IEEE standard 754.

Fig. 2 depicts a prior art floating point addition pipeline;

Fig. 3 depicts a prior art floating point addition pipeline having far and close data paths;

Fig. 4 is a block diagram of a microprocessor according to one embodiment of the present invention;

Fig. 5 is a block diagram of an execution unit having an add/subtract pipeline according to one embodiment of the present invention;

Fig. 6 is a block diagram of one embodiment of a far data path within the add/subtract pipeline of Fig. 5;

Fig. 7 is a block diagram of one embodiment of a multiplexer-inverter unit within the far data path of Fig. 6;

Fig. 8 is a block diagram of one embodiment of an adder unit within the far data path of Fig. 6;

Fig. 9 is a block diagram of one embodiment of a selection unit within the far data path of Fig. 6;

Figs. 10A-H are examples of addition and subtraction performed within the far data path of Fig. 6;

Fig. 11 is a block diagram of one embodiment of a multiplexer-shift unit within the c data path of Fig. 6;

Fig. 12 is a block diagram of one embodiment of a close data path within the add/subtract pipeline of Fig. 5;

Fig. 13 is a block diagram of one embodiment of a shift-swap unit within the close data path of Fig. 12;

Fig. 14 is a block diagram of one embodiment of an adder unit within the close data path of Fig. 12;

Fig. 15 is a block diagram of one embodiment of a selection unit 730 within the close data path of Fig. 12;

Figs. 16A-G are examples of subtraction performed within the close data path of Fig. 12;

Fig. 17 is a block diagram of one embodiment of a multiplexer-inverter unit 740 within the close data path of Fig. 12;

Fig. 18 is a block diagram of one embodiment of a left shift unit 750 within the close data path of Fig. 12;

Fig. 19 is a block diagram of one embodiment of a result multiplexer unit 250 within the close data path of Fig. 12;

Fig. 20 is a block diagram of a prior art leading 0/1 prediction unit 1400;

Fig. 21 is a block diagram of a prior art TGZ generation unit within prediction unit 1400 of Fig. 20;

Figs. 22A-C are examples of how T-G-Z prediction strings may be utilized to perform leading 0/1 prediction;

Fig. 23 is a logic diagram of a prediction unit configured to form both leading 0 and 1 prediction strings;

Fig. 24 is a prior art simplification of a TGZ generation unit for operands A and B, where A > B;

Fig. 25 illustrates the derivation of a simplified leading 1 prediction units in which exponent E_A of a first operand is one greater than exponent E_B of a second operand;

Fig. 26 is a block diagram of one embodiment of an improved leading 1 prediction unit for which $E_A=E_B+1$;

Figs. 27A-B depict floating point numbers and converted integer equivalents according to one embodiment of the present invention;

Fig. 28 is a block diagram of one embodiment of a far data path 2300 which is configured to perform floating point to integer (f2i) conversions;

Fig. 29 is a block diagram of one embodiment of a multiplexer inverter unit 2330 within far data path 2300 of Fig. 28;

Fig. 30 is a block diagram of one embodiment of a result multiplexer unit 2500 within far data path 2300 of Fig. 28;

Figs. 31A-B depict integer numbers and converted floating point equivalents according to one embodiment of the present invention;

Fig. 32 is a block diagram of one embodiment of a close data path 2600 which is configured to perform integer-to-floating point (i2f) conversions;

Fig. 33 is a block diagram of one embodiment of a shift-swap unit 2610 within close data path 2600 of Fig. 32;

Fig. 34 is a block diagram of one embodiment of a multiplexer-inverter unit 2640 within close data path 2600 of Fig. 32;

Fig. 35 is a block diagram of one embodiment of an exponent within close data path 2600 of Fig. 32;

Fig. 36 is a block diagram of one embodiment of an execution unit within microprocessor 100 which includes a plurality of add/subtract pipelines;

Fig. 37A depicts the format of a vectored floating point addition instruction according to one embodiment of the invention;

Fig. 37B depicts pseudocode for the vectored floating point addition instruction of Fig. 37A;

Fig. 38A depicts the format of a vectored floating point subtraction instruction according to one embodiment of the invention;

Fig. 38B depicts pseudocode for the vectored floating point subtraction instruction of Fig. 38A;

Fig. 39A depicts the format of a vectored floating point-to-integer conversion instruction according to one embodiment of the invention;

Fig. 39B depicts pseudocode for the vectored floating point-to-integer conversion instruction of Fig. 39A;

Fig. 39C is a table listing output values for various inputs to the vectored floating point-to-integer conversion instruction of Fig. 39A;

Fig. 40A depicts the format of a vectored floating point-to-integer conversion instruction according to an alternate embodiment of the invention;

Fig. 40B depicts pseudocode for the vectored floating point-to-integer conversion instruction of Fig. 40A;

Fig. 40C is a table listing output values for various inputs to the vectored floating point-to-integer conversion instruction of Fig. 40A;

Fig. 41A depicts the format of a vectored integer-to-floating point conversion instruction according to one embodiment of the invention;

Fig. 41B depicts pseudocode for the vectored integer-to-floating point conversion instruction of Fig. 41A;

Fig. 42A depicts the format of a vectored integer-to-floating point conversion instruction according to an alternate embodiment of the invention;

Fig. 42B depicts pseudocode for the vectored integer-to-floating point conversion instruction of Fig. 42A;

Fig. 43A depicts the format of a vectored floating point accumulate instruction according to one embodiment of the invention;

Fig. 43B depicts pseudocode for the vectored floating point accumulate instruction of Fig. 43A;

Fig. 44A depicts the format of a vectored floating point reverse subtract instruction according to one embodiment of the invention;

Fig. 44B depicts pseudocode for the vectored floating point reverse subtract instruction of Fig. 44A;

Fig. 45A depicts the format of a vectored floating point maximum value instruction according to one embodiment of the invention;

Fig. 45B depicts pseudocode for the vectored floating point maximum value instruction of Fig. 45A;

Fig. 45C is a table listing output values for various inputs to the vectored floating point maximum value instruction of Fig. 45A;

Fig. 46A depicts the format of a vectored floating point minimum value instruction according to one embodiment of the invention;

Fig. 46B depicts pseudocode for the vectored floating point minimum value instruction of Fig. 46A;

Fig. 46C is a table listing output values for various inputs to the vectored floating point minimum value instruction of Fig. 46A;

Fig. 47A depicts the format of a vectored floating point equality comparison instruction according to one embodiment of the invention;

Fig. 47B depicts pseudocode for the vectored floating point equality comparison instruction of Fig. 47A;

Fig. 47C is a table listing output values for various inputs to the vectored floating point equality comparison instruction of Fig. 47A;

Fig. 48A depicts the format of a vectored floating point greater than comparison instruction according to one embodiment of the invention;

Fig. 48B depicts pseudocode for the vectored floating point greater than comparison instruction of Fig. 48A;

Fig. 48C is a table listing output values for various inputs to the vectored floating point greater than comparison instruction of Fig. 48A;

Fig. 49A depicts the format of a vectored floating point greater than or equal to comparison instruction according to one embodiment of the invention;

Fig. 49B depicts pseudocode for the vectored floating point greater than or equal to comparison instruction of Fig. 49A;

Fig. 49C is a table listing output values for various inputs to the vectored floating point greater than or equal to comparison instruction of Fig. 49A;

Fig. 50 is a block diagram of one embodiment of an execution unit 136C/D according to one embodiment of the invention which is configured to executed the instructions of Figs. 37-49; and

Fig. 51 is a block diagram of one embodiment of a computer system which includes microprocessor 100.

Fig. 52 is a block diagram of a microprocessor which configured according to one embodiment of the present invention;

Fig. 53 is a graph depicting a portion of a function f(x) which is partitioned for use with a prior art naive look-up table;

Fig. 54 is a prior art naive look-up table usable in conjunction with the function partitioned according to Fig.52;

Fig. 55 is a graph depicting a portion of a function f(x) which is partitioned for use with a prior art bipartite look-up table;

Fig. 56 is a prior art bipartite look-up table usable in conjunction with the function partitioned according to Fig. 55;

Fig. 57 is a graph depicting a portion of a function f(x) which is partitioned for use with a bipartite look-up table according to one embodiment of the present invention;

Fig. 58 is a bipartite look-up table usable in conjunction with the function partitioned according to Fig. 57;

Fig. 59 depicts one format for an input value to a bipartite look-up in accordance with one embodiment of the present invention;

Fig. 60A illustrates a look-up table input value according to the format of Fig. 59 in one embodiment of the present invention;

Fig. 60B depicts the mantissa portion of a look-up table input value for the reciprocal function;

Fig. 60C depicts a base table index for a bipartite look-up table for the reciprocal function, according to one embodiment of the present invention;

Fig. 60D depicts a difference table index for a bipartite look-up table for the reciprocal function, according to one embodiment of the present invention;

Fig. 61A depicts the mantissa portion of a look-up table input value for the reciprocal square root function;

Fig. 61B depicts a base table index for a bipartite look-up table for the reciprocal square root function, according to one embodiment of the present invention;

Fig. 61C depicts a difference table index for a bipartite look-up table for the reciprocal square root function, according to one embodiment of the present invention;

Fig. 62 is a bipartite look-up table for the reciprocal and reciprocal square root functions according to one embodiment of the present invention;

Fig. 63 is one embodiment of an address control unit within the bipartite look-up table of Fig. 62;

Fig. 64A is a graph depicting a prior art midpoint calculation for a bipartite look-up table;

Fig. 64B is a graph depicting a midpoint calculation for a bipartite look-up table according to one embodiment of the present invention;

Fig. 65A is a flowchart depicting a method for computation of difference table entries for a bipartite look-up table according to one embodiment of the present invention;

Fig. 65B is a graph depicting difference value averaging over a portion of a function f(x) partitioned for use with a bipartite look-up table according to one embodiment of the present invention;

Figs. 66A-B are graphs comparing table output values for a portion of a function f(x) to computed midpoint values for the function portion;

Figs. 66C-D are graphs comparing table outputs with adjusted base values for a portion of a function f(x) to computed midpoint values for the function portion; and

Fig. 67 is a flowchart depicting a method for computation of base table entries for a bipartite look-up table according to one embodiment of the present invention.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE INVENTION

Turning now to Fig. 4, a block diagram of one embodiment of a microprocessor 100 is shown. As depicted, microprocessor 100 includes a predecode logic block 112 coupled to an instruction cache 114 and a predecode cache 115. Caches 114 and 115 also include an instruction TLB 116. A cache controller 118 is coupled to predecode block 112, instruction cache 114, and predecode cache 115. Controller 118 is additionally coupled to a bus interface unit 124, a level-one data cache 126 (which includes a data TLB 128), and an L2 cache 140. Microprocessor 100 further includes a decode unit 120, which receives instructions from instruction cache 114 and predecode data from cache 115. This information is forwarded to execution engine 130 in accordance with input received from a branch logic unit 122.

Execution engine 130 includes a scheduler buffer 132 coupled to receive input from decode unit 120. Scheduler buffer 132 is coupled to convey decoded instructions to a plurality of execution units 136A-E in accordance with input received from an instruction control unit 134. Execution units 136A-E include a load unit 136A, a store unit 136B, an integer/multimedia X unit 136C, an integer/multimedia Y unit 136D, and a floating point unit 136E. Load unit 136A receives input from data cache 126, while store unit 136B interfaces with data cache 126 via a store queue 138. Blocks referred to herein with a reference number followed by a letter will be collectively referred to by the reference number alone. For example, execution units 136A-E will be collectively referred to as execution units 136.

In one embodiment, instruction cache 114 is organized as sectors, with each sector including two 32-byte cache lines.. The two cache lines of a sector share a common tag but have separate state bits that track the status of the line. Accordingly, two forms of cache misses (and associated cache fills) may take place: sector replacement and cache line replacement. In the case of sector replacement, the miss is due to a tag mismatch in instruction cache 114, with the required cache line being supplied by external memory via bus interface unit 124. The cache line within the sector that is not needed is then marked invalid. In the case of a cache line replacement, the tag matches the requested address, but the line is marked as invalid. The required cache line is supplied by external memory, but, unlike the sector replacement case, the cache line within the sector that was not requested remains in the same state. In alternate embodiments, other organizations for instruction cache 114 may be utilized, as well as various replacement policies.

Microprocessor 100 performs prefetching only in the case of sector replacements in one embodiment. During sector replacement, the required cache line is filled. If this required cache line is in the first half of the

sector, the other cache line in the sector is prefetched. If this required cache line is in the second half of the sector, no prefetching is performed. It is noted that other prefetching methodologies may be employed in different embodiments of microprocessor 100.

When cache lines of instruction data are retrieved from external memory by bus interface unit 124, this data is conveyed to predecode logic block 112. In one embodiment, the instructions processed by microprocessor 100 and stored in cache 114 are variable-length (e.g., the x86 instruction set). Because decode of variable-length instructions is particularly complex, predecode logic 112 is configured to provide additional information to be stored in predecode cache 115 to aid during decode. In one embodiment, predecode logic 112 generates predecode bits for each byte in instruction cache 114 which indicate the number of bytes to the start of the next variable-length instruction. These predecode bits are stored in predecode cache 115 and are passed to decode unit 120 when instruction bytes are requested from cache 114.

Instruction cache 114 is implemented as a 32Kbyte, two-way set associative, writeback cache in one embodiment of microprocessor 100. The cache line size is 32 bytes in this embodiment. Cache 114 also includes a TLB 116, which includes 64 entries used to translate linear addresses to physical addresses. Many other variations of instruction cache 114 and TLB 116 are possible in other embodiments.

Instruction fetch addresses are supplied by cache controller 118 to instruction cache 114. In one embodiment, up to 16 bytes per clock cycle may be fetched from cache 114. The fetched information is placed into an instruction buffer that feeds into decode unit 120. In one embodiment of microprocessor 100, fetching may occur along a single execution stream with seven outstanding branches taken.

In one embodiment, the instruction fetch logic within cache controller 118 is capable of retrieving any 16 contiguous instruction bytes within a 32-byte boundary of cache 114. There is no additional penalty when the 16 bytes cross a cache line boundary. Instructions are loaded into the instruction buffer as the current instructions are consumed by decode unit 120. (Predecode data from cache 115 is also loaded into the instruction buffer as well). Other configurations of cache controller 118 are possible in other embodiments.

Decode logic 120 is configured to decode multiple instructions per processor clock cycle. In one embodiment, decode unit 120 accepts instruction and predecode bytes from the instruction buffer (in x86 format), locates actual instruction boundaries, and generates corresponding "RISC ops". RISC ops are fixed-format internal instructions, most of which are executable by microprocessor 100 in a single clock cycle. RISC ops are combined to form every function of the x86 instruction set in one embodiment of microprocessor 100.

Microprocessor 100 uses a combination of decoders to convert x86 instructions into RISC ops. The hardware includes three sets of decoders: two parallel short decoders, one long decoder, and one vectoring decoder. The parallel short decoders translate the most commonly-used x86 instructions (moves, shifts, branches, etc.) into zero, one, or two RISC ops each. The short decoders only operate on x86 instructions that are up to seven bytes long. In addition, they are configured to decode up to two x86 instructions per clock cycle. The commonly-used x86 instructions which are greater than seven bytes long, as well as those semi-commonly-used instructions are up to seven bytes long, are handled by the long decoder.

The long decoder in decode unit 120 only performs one decode per clock cycle, and generates up to four RISC ops. All other translations (complex instructions, interrupts, etc.) are handled by a combination of the vector decoder and RISC op sequences fetched from an on-chip ROM. For complex operations, the vector

decoder logic provides the first set of RISC ops and an initial address to a sequence of further RISC ops. The RISC ops fetched from the on-chip ROM are of the same type that are generated by the hardware decoders.

In one embodiment, decode unit 120 generates a group of four RISC ops each clock cycle. For clock cycles in which four RISC ops cannot be generated, decode unit 120 places RISC NOP operations in the remaining slots of the grouping. These groupings of RISC ops (and possible NOPs) are then conveyed to scheduler buffer 132.

It is noted that in another embodiment, an instruction format other than x86 may be stored in instruction cache 114 and subsequently decoded by decode unit 120.

Instruction control unit 134 contains the logic necessary to manage out-of-order execution of instructions stored in scheduler buffer 132. Instruction control unit 134 also manages data forwarding, register renaming, simultaneous issue and retirement of RISC ops, and speculative execution. In one embodiment, scheduler buffer 132 holds up to 24 RISC ops at one time, equating to a maximum of 12 x86 instructions. When possible, instruction control unit 134 may simultaneously issue (from buffer 132) a RISC op to any available one of execution units 136. In total, control unit 134 may issue up to six and retire up to four RISC ops per clock cycle in one embodiment.

In one embodiment, microprocessor 10 includes five execution units (136A-E). Load unit 136A and store unit 136B are two-staged pipelined designs. Store unit 136B performs data memory and register writes which are available for loading after one clock cycle. Load unit 136A performs memory reads. The data from these reads is available after two clock cycles. Load and store units are possible in other embodiments with varying latencies.

Execution unit 136C is configured, in one embodiment, to perform all fixed point ALU operations, as well as multiplies, divides (both signed and unsigned), shifts, and rotates. Execution unit 136D, in contrast, is configured to perform basic word and double word ALU operations (ADD, AND, CMP, etc.). Additionally, units 136C-D are configured to accelerate performance of software written using multimedia instructions. Applications that can take advantage of multimedia instructions include graphics, video and audio compression and decompression, speech recognition, and telephony. Accordingly, units 136C-D are configured to execute multimedia instructions in a single clock cycle in one embodiment. Many of these instructions are designed to perform the same operation of multiple sets of data at once (vector processing). In one embodiment, these multimedia instructions include both vectored fixed point and vectored floating point instructions.

Execution unit 136E contains an IEEE 754-compatible floating point unit designed to accelerate the performance of software which utilizes the x86 instruction set. Floating point software is typically written to manipulate numbers that are either very large or small, require a great deal of precision, or result from complex mathematical operations such as transcendentals. Floating point unit includes an adder unit, a multiplier unit, and a divide/square root unit. In one embodiment, these low-latency units are configured to execute floating point instructions in as few as two clock cycles.

Branch resolution unit 135 is separate from branch prediction logic 122 in that it resolves conditional branches such as JCC and LOOP after the branch condition has been evaluated. Branch resolution unit 135 allows efficient speculative execution, enabling microprocessor 100 to execute instructions beyond conditional

branches before knowing whether the branch prediction was correct. As described above, microprocessor 100 is configured to handle up to seven outstanding branches in one embodiment.

Branch prediction logic 122, coupled to decode unit 120, is configured to increase the accuracy with which conditional branches are predicted in microprocessor 100. Ten to twenty percent of the instructions in typical applications include conditional branches. Branch prediction logic 122 is configured to handle this type of program behavior and its negative effects on instruction execution, such as stalls due to delayed instruction fetching. In one embodiment, branch prediction logic 122 includes an 8192-entry branch history table, a 16-entry by 16 byte branch target cache, and a 16-entry return address stack.

Branch prediction logic 122 implements a two-level adaptive history algorithm using the branch history table. This table stores executed branch information, predicts individual branches, and predicts behavior of groups of branches. In one embodiment, the branch history table does not store predicted target addresses in order to save space. These addresses are instead calculated on-the-fly during the decode stage.

To avoid a clock cycle penalty for a cache fetch when a branch is predicted taken, a branch target cache within branch logic 122 supplies the first 16 bytes at that address directly to the instruction buffer (if a hit occurs in the branch target cache). In one embodiment, this branch prediction logic achieves branch prediction rates of over 95%.

Branch logic 122 also includes special circuitry designed to optimize the CALL and RET instructions. This circuitry allows the address of the next instruction following the CALL instruction in memory to be pushed onto a return address stack. When microprocessor 100 encounters a RET instruction, branch logic 22 pops this address from the return stack and begins fetching.

Like instruction cache 114, L1 data cache 126 is also organized as two-way set associative 32Kbyte storage. In one embodiment, data TLB 128 includes 128 entries used to translate linear to physical addresses. Like instruction cache 114, L1 data cache 126 is also sectored. Data cache 126 implements a MESI (modified-exclusive-shared-invalid) protocol to track cache line status, although other variations are also possible. In order to maximize cache hit rates, microprocessor 100 also includes on-chip L2 cache 140 within the memory sub-system.

Turning now to Fig. 5, a block diagram of a portion of an execution unit 136C/D is depicted. The "C/D" denotes that the execution unit shown in Fig. 5 is representative of both execution units 136C and 136D. This means of reference is also used below to describe other embodiments execution units 136C-D. As shown, execution unit 136C/D includes an input unit 210 which receives an add/subtract indication 202 and operands 204A-B. Input unit 210 is coupled an add/subtract pipeline 220, which includes a far data path 230 and a close data path 240. Far data path 230 and close data path 240 receive inputs from input unit 210 and generate far path result 232 and close path result 242, respectively, which are conveyed to a result multiplexer unit 250. Far data path 230 also conveys a select signal to multiplexer unit 250 in one embodiment. In this embodiment, the select signal is usable to select either far path result 232 or close path result 242 to be conveyed as result value 252, which is the output of add/subtract pipeline 220.

Input unit 210 receives the operand data, and conveys sufficient information to far data path 230 and close data path 240 to perform the add or subtract operation. In one embodiment, add/subtract indication 202 is indicative of the operation specified by the opcode of a particular floating point arithmetic instruction. That is,

add/subtract indication 202 corresponds to the opcode of an instruction being processed by unit 136C/D (a logic 0 may indicate an add opcode and a logic 1 a subtract opcode in one embodiment). Operands 204 are floating point numbers having sign, exponent, and mantissa portions according to a predetermined floating point format (such as IEEE standard 754). If add/subtract indication 202 corresponds to an opcode add/subtract value, input unit 210 may be configured to make a determination whether effective addition or subtraction is occurring. (As described above, an subtract opcode value may effectively be an addition operand depending on the signs of operands 204). In one embodiment, input unit 210 determines whether inputs 202 and 204 represent effective addition or subtraction, and conveys outputs to far data path 230 and close data path 240. In an alternate embodiment, the determination of effective addition or subtraction is made prior to conveyance to unit 136C/D. Add/subtract indication 202 is thus reflective of either effective addition or subtraction, and sign bits of incoming operands 204 are adjusted accordingly. In yet another embodiment, the effective addition/subtraction determination may be made separately within far data path 230 and close data path 240.

The format of the outputs of input unit 210 depends upon the format of unit 210 inputs and also the configuration of far data path 230 and close data path 240. In one embodiment, unit 210 conveys the full sign, exponent, and mantissa values (S_A, S_B, E_A, E_B, M_A, and M_B) of operands 204 to far data path 230, while conveying S_A, S_B, M_A, M_B, and two least significant bits of both E_A and E_B to close data path 240. As will be described the two least significant exponents bits are used for speculative determination of exponent difference (instead of a full subtract). In other embodiments of add/subtract pipeline 220, far data path 230 and close data path 240 may receive input data of varying formats.

Far data path 230 is configured to perform addition operations, as well as subtraction operations for operands having absolute exponent difference E_{diff} which is greater than 1. Close data path 240, on the other hand, is configured to perform subtraction operations on operands for which $E_{diff} \leq 1$. As will be described below, close data path 240 includes a selection unit which is configured to provide improved performance over prior art pipelines such as pipelines 10 and 30 described above.

Far data path 230 and close data path 240 generate far path result 232 and close path result 242, respectively, which are both conveyed to result multiplexer unit 250. As shown, far data path also generates a select signal for unit 250, which is usable to select either input 232 or 242 as result value 252. In alternate embodiments of add/subtract pipeline 220, the select for multiplexer unit 250 may generated differently.

Turning now to Fig. 6, a block diagram of far data path 230 is depicted. As shown, far data path 230 receives an add/subtract indication, full exponent values (E_A and E_B), and full mantissa values (M_A and M_B) from input unit 210 in one embodiment. In the embodiment shown, data path 230 also receives sign bits S_A and S_B, although they are not depicted in Fig. 6 for simplicity and clarity.

Far data path 230 includes exponent difference calculation units 310A-B, which receive input exponent values E_A and E_B. Units 310 are coupled to right shift units 314A-B, which receive mantissa values M_A and M_B, respectively. Shift units 314 are also coupled to multiplexer-inverter unit 330 and logic unit 320 (referred to as "GRS" logic because unit 320 stores the guard (G), round (R), and sticky (S) bits shifted out in units 314). Multiplexer-inverter unit 330, in response to receiving shifted (316A-B) and unshifted versions of M_A and M_B, conveys a pair of operands (332A-B) to an adder unit 340. Adder unit 340, in turn, generates a pair of outputs 342A and 342B, which are conveyed to multiplexer-shift unit 360. Adder unit 340 is additionally coupled to a

selection unit 350, which generates a select signal for multiplexer-shift unit 360. Selection unit 350 also receives inputs from exponent unit 310 and GRS logic unit 320 in addition to values from adder unit 340. In response to select signal 352 conveyed from selection unit 350, multiplexer shift unit 360 conveys a mantissa value which, when coupled with an adjusted exponent value conveyed from an exponent adjust unit 370, is conveyed as far path result 232 to result multiplexer unit 250. Exponent adjust unit 370 receives the largest input exponent 309 (which is equal to max(E_A, E_B)) from an exponent comparator unit 308 coupled to receive E_A and E_B. Exponent 309 is additionally conveyed to close data path 240 for exponent calculations as is described below.

As shown in Fig. 6, exponent difference unit 310A is coupled to receive full exponent values E_A and E_B. Unit 310A is configured to compute the difference E_B-E_A and convey the resulting shift amount, 312A, to right shift unit 314A. Exponent difference unit 310B also receives full exponent values E_A and E_B, but is configured to compute the difference E_A-E_B, which is conveyed as shift amount 312B to right shift unit 314B. In this embodiment, unless E_A=E_B, one of result 312 is negative (and therefore ultimately discarded by pipeline 220). An embodiment is also contemplated in which only one right shift unit 314 is provided; however, additional multiplexer logic may be needed to convey the proper mantissa value to the single shift unit. By providing two shift units 314, the performance of far data path 230 is increased.

Shift amount 312A, in one embodiment, is conveyed to a final select generation unit 311, along with add/subtract indication 202. Unit 311, in turn, generates an exponent difference select signal 313 to be conveyed to result multiplexer unit 250. The signal 313 generated by unit 310 is indicative of either far path result 232 or close path result 242. Signal 313 may thus be used by result multiplexer unit 250 to select either result 232 or result 242 as result value 252. If add/subtract indication 202 specifies an add operation, signal 313 is generated to be indicative of far path result 232. Similarly, if add/subtract indication 202 specifies a subtract operation and E_{diff} (corresponding to the absolute value of shift amount 312A) is greater than one, signal 313 is also generated to be indicative of far path result 232. Conversely, if add/subtract indication 202 specifies a subtract operation and E_{diff} is 0 or 1, signal 313 is generated to be indicative of close path result 242. In one embodiment, signal 313 may be used to cancel the far path result if E_{diff} indicates result 242. E_{diff} is also conveyed to selection unit 350 in one embodiment, as will be described below.

Right shift units 314A-B generate shift outputs 316A-B, respectively, according to shift amounts 312A-B. These shift outputs are then conveyed to multiplexer-inverter unit 330. Unit 330 is also coupled to receive add/subtract indication from input unit 210 and the sign bit of shift amount 312A. In one embodiment, multiplexer-inverter unit 330 is configured to swap operands 316A and 316B if operand 316B is determined to be greater than operand 316A. This determination may be made in one embodiment from the sign bit of shift amount 312A (or 312B). Additionally, unit 330 is configured to invert the smaller operand if subtraction is indicated by input unit 210. The outputs of unit 330 are conveyed to adder unit 340 as adder inputs 332A-B.

GRS logic unit 320 receives values which are right-shifted out of units 314A-B. After shift amounts 312 are applied to values in shift units 314, GRS logic unit 320 generates guard, round, and sticky bits corresponding to the smaller mantissa value. As shown, these bit values are forwarded to selection unit 350 for the rounding computation.

Adder unit 340 receives adder inputs 332A-B and generates a pair of output values 342A-B. Output 342A corresponds to the sum of input values 332 (sum), while output 342B corresponds to output 342A plus one (sum+1). Adder unit 340 also conveys a plurality of signals to selection unit 350, which generates and conveys select signal 352 to multiplexer-shift unit 360. Select signal 352 is usable to select either adder output 342A-B to be conveyed as the mantissa portion of far path result 232. By selecting either sum or sum+1 as the output of multiplexer-shift unit 360, the addition result may effectively be rounded according to the IEEE round-to-nearest mode.

In one embodiment, the exponent portion of far path result 232 is generated by exponent adjustment unit 370. Unit 370 generates the adjusted exponent from the original larger exponent value (either E_A or E_B) and an indication of whether the adder output is normalized. The output of unit 370 is conveyed along with the output of unit 360 as far path result 232.

Turning now to Fig. 7, a block diagram of multiplexer-inverter unit 330 is depicted. Unit 330 includes a control unit 331 which receives shift amount 312A from exponent difference calculation unit 310A. Multiplexer-inverter unit 330 also includes a pair of input multiplexers 334A-B. Input multiplexer 334A receives unshifted mantissa values M_A and M_B, while multiplexer 334B receives shifted outputs 316A-B. In one embodiment, the inputs to multiplexers 334 are configured such that control unit 331 may route a single control signal 333 to both multiplexer 334A and 334B. Additionally, the output of multiplexer 334B is inverted by an inverter 336 if a subtract operation is indicated by signal 202. If a subtract is indicated, a bit-inverted (one's complement) version of the output of multiplexer 334B is conveyed to adder 340 as adder input 342B. If an add operation is indicated by signal 202, inverter 336 is not enabled, and the output of multiplexer 334B is conveyed to adder unit 340 in non-inverted form.

Turning now to Fig. 8, a block diagram of one embodiment of adder unit 340 is depicted. Adder unit 340 includes adders 400A and 400B, each coupled to receive adder inputs 332A-B. Adder 400A is configured to generate adder output 342A (sum), while adder 400B is configured to generate adder output 342B (sum+1).

As shown, adders 400A and 400B are each coupled to receive the sign and mantissa bits of operands 204A-B. In one embodiment, adders 400A and 400B are identical except that adder 400B has a carry in (C_{LSB}) value of 1, while, for adder 400A, C_{LSB}=0. It is contemplated that adders 400 may be implemented using a variety of known adder types. For example, adders 400 may be implemented as ripple-carry adders, carry lookahead adders, carry-select adders, etc. Furthermore, adders 400 may combine features of different adder types. In one embodiment, adders 400 compute the upper n/2 bits of their respective results in two different ways: that the carry in from the lower n/2 bits is 0, and that the carry in from the lower n/2 bits is 1. The use of Ling-style pseudo-carry may also be utilized in the lower n/2 bits to further reduce fan-in and gate delay. In yet another embodiment, adder unit 340 may be implemented with just a single adder. This may be accomplished by recognizing that many of the terms computed in adders 400A-B are shared. Accordingly, both sum and sum+1 may be produced by a single adder. Although such an adder is larger (in terms of chip real estate) than either of adders 400, the single adder represents a significant space savings vis-a-vis the two adder configuration of Fig. 8.

As will be described below, the most significant bit of the output of adder 400A (S_{MSB}) is used by selection unit 350 to generate select signal 352. The faster select signal 352 is generated, then, the faster result

value 252 can be computed. Accordingly, in the embodiment shown in Fig. 8, S_{MSB} is generated in selection unit 350 concurrently with the MSB computation performed in adder 400A. To facilitate this operation, A_{MSB}, B_{MSB}, and C_{MSB} (the carry in to adder block 402B which generates S_{MSB}) are all conveyed to selection unit 350. By conveying the inputs to adder block 402B to selection unit 350 in parallel, the output of selection unit 350 may be generated more quickly, enhancing the performance of far data path 230. The two least significant bits of adder output 342A (S_{LSB+1} and S_{LSB}) are also conveyed to selection unit 350. In one embodiment, these values are not generated in parallel in unit 350 (in the manner of S_{MSB}) since the least significant bits are available relatively early in the addition operation (in contrast to more significant bits such as S_{MSB}).

As noted above, adder 400B operates similarly to adder 400A, except that carry in value 404B is a logical one. Since the carry in value (404A) for adder 400A is a logical zero, adder 400B generates a result equal to the output of adder 400A plus one. As will be described below, by generating the values (sum) and (sum+1) for a given pair of operands, the IEEE round to nearest mode may be effectuated by selecting one of the two values.

Turning now to Fig. 9, a block diagram of selection unit 350 is shown in one embodiment of far data path 230. The general operation of selection unit 350 is described first, followed by examples of far path computations.

As shown, selection unit 350 receives a plurality of inputs from adder unit 340. These inputs include, in one embodiment, the inputs to adder 400A block 402B (A_{MSB}, B_{MSB}, and C_{MSB}), the next-to-least significant bit (N) of adder output 342A, the least significant bit (L) of adder output 342B, and the guard (G), round (R), and sticky (S) bits from GRS logic unit 320. A logical-OR of the round and sticky bits, S_1, is produced by logic gate 502. Bit S_1 is used for calculations in which R is not explicitly needed. Selection unit 350 also includes a selection logic block 510 which includes selection sub-blocks 510A-D. In response to the inputs received from units 320 and 340, sub-blocks 510A-D generate respective select signals 512A-D. Select signals 512 are conveyed to a far path multiplexer 520, which also receives control signals including add/subtract indication 202, S_{MSB} signal 534, and C_S signal 536. S_{MSB} signal 534 is conveyed from a multiplexer 530A, while C_S is conveyed from a multiplexer 530B. In response to these control signals, multiplexer 520 conveys one of select signals 512 as far path select signal 352 to multiplexer-shift unit 360.

As described above, adder unit 340 is configured to generate sum and sum+1 for operands 204A and 240B. Selection unit 350 is configured to generate far path select signal 352 such that the sum/sum+1 is a) corrected for one's complement subtraction and b) rounded correctly according to the IEEE round-to-nearest mode. In general, a number generated by one's complement subtraction must have 1 added in at the LSB to produce a correct result. Depending on the state of the G, R, and S bits, however, such correction may or may not be needed. With respect to rounding, sum+1 is selected in some instances to provide a result which is rounded to the next highest number. Depending on various factors (type of operation, normalization of output 342A), sum or sum+1 is selected using different selection equations. Accordingly, selection sub-blocks 510A-D speculatively calculate selection values for all possible scenarios. These selection values are conveyed to multiplexer 520 as select signals 512A-D. Control signals 302, 534, and 536 indicate which of the predicted select signals 512 is valid, conveying one of signals 512 as far path select signal 352.

Turning now to Figs. 10A-B, examples of addition accurately predicted by selection sub-block 510A are shown. Since sub-block 510A only predicts for addition, selection of sum+1 is used for rounding purposes only. Fig. 10A depicts an addition example 550A in which sum is selected. Rounding is not performed since $G(L+S_1)$ is not true. Conversely, Fig. 10B depicts an addition example 550B in which sum+1 is selected. Because G and S_1 are set, the result is closer to 1.01011 than to 1.01010. Accordingly, sum+1 (1.01011) is selected.

Turning now to Figs. 10C-10D, examples of addition accurately predicted by selection sub-block 510B are shown. Since sub-block 510B only predicts for addition, selection of sum+1 is used for rounding purposes only. The examples shown in Figs. 10C-D are similar to those shown in Figs. 10A-B except that overflow conditions are present in examples 550C-D shown in Figs. 10C-D. Accordingly, the equation for selecting sum+1 is slightly different than for selection sub-block 510A. Fig. 10C depicts an addition example 550C in which sum is selected. Conversely, Fig. 10D depicts an addition example 550D in which sum+1 is selected, effectively rounding up the result (after a 1-bit right shift to correct for overflow). Selection sub-block 510B selects sum+1 according to the equation $L(N+G+S_1)$.

Turning now to Figs. 10E-F, examples of addition accurately predicted by selection sub-block 510C are shown. Since sub-block 510C is used to predict selection for subtraction operations which have properly normalized results , selection of sum+1 is performed to correct for one's complement subtraction and for rounding purposes. As shown in example 550E, sum is indicated by select signal 512C since the guard and sticky bits are set before the subtract (ensuring that the result of the subtraction is closer to sum than sum+1). Conversely, in example 550F, the guard and sticky bits are both zero. Accordingly, a one-bit addition to the LSB is needed; therefore, sum+1 is selected. Generally speaking, selection sub-block 510C selects sum+1 according to the equation $G'+LS_1'$, where G' and S_1' represent the complements of the G and S_1 bits.

Turning now to Figs. 10G-H, examples of addition accurately predicted by selection sub-block 510D are shown. Since sub-block 510D is used to predict selection for subtract operations which require a 1-bit left shift of the result, selection of sum+1 is performed for both one's complement correction and rounding. In example 550G, sum is chosen as the result since both the guard and round bits are set before the subtract (ensuring that the result of the subtraction is closer to sum than sum+1). For this particular example, a zero is shifted into the LSB when the result is normalized. (In other examples, a one may be shifted in). In example 550H, both the guard and round bits are zero, which causes the result of the subtraction to be closer to sum+1 than sum. Accordingly, sum+1 is selected. A zero is shifted in at the LSB. Generally speaking, selection sub-block 510D selects sum+1 according to the equation $G'(R'+S')$, while the shift value is generated according to the equation $GR'+G'RS$.

It is noted that other embodiments of selection unit 350 are also possible. For example, in selection sub-blocks 510C and 510D, the guard and round bit inputs may be inverted if the sticky bit is set, resulting in different rounding equations. Various other modifications to the selection logic are possible as well.

Turning now to Fig. 11, a block diagram of multiplexer-shift unit 360 is depicted in one embodiment of far data path 230. As shown, multiplexer-shift unit 360 is coupled to receive adder outputs 342A-B and shift value 514. A concatenation unit 610 receives outputs 342 and shift value 514, and conveys shifted multiplexer outputs 604A-D to multiplexer 600. Multiplexer 600 receives signals 352 (far path select signal), 534 (S_{MSB}),

and 536 (C_{MSB}) as control inputs. In response to these control signals, multiplexer 600 selects one of signals 342 or 604 as far path mantissa result 612. The exponent portion of far path result 232 is conveyed by exponent adjustment unit 370, which adjusts the original larger exponent value, in one embodiment, by the amount of normalization (or correction for overflow) required by the result.

As shown, multiplexer 600 includes three groups of inputs, denoted as A, B, and C. Inputs A0 and A1 are adder outputs 342, representing sum and sum+1. Inputs B0 and B1 (signals 604A-B), on the other hand, represent adder outputs 342 adjusted for overflow (a '0' is routed as the MSB by concatenation unit 610). Finally, inputs C0 and C1 represent adder outputs 342 after a one-bit left shift. Concatenation unit 610 utilizes the shift value conveyed from selection sub-block 510D to append as the LSB of the conveyed outputs 604C-D.

In one embodiment, signals 534 and 536 are usable to determine whether adder output 342A is normalized properly (input group A), has an overflow condition (input group B), or requires a one-bit left shift (input group C). Far path select signal 352 is then usable to determine which input within the selected input group is to be conveyed as far path mantissa result 612.

Turning now to Fig. 12, a block diagram of one embodiment of close data path 240 is depicted. As described above, close data path 240 is configured to perform effective subtraction operations for operands having an absolute exponent difference of 0 or 1. Subtraction operations with operands having other absolute exponent difference values (and all addition operations) are handled as described above in far data path 230.

As shown, close data path 240 receives a variety of inputs from input unit 210. Close data path 240 includes an exponent prediction unit 704, which receives the two least significant exponent bits of exponents E_A and E_B. In one embodiment, exponent prediction unit 704 generates a prediction 706 regarding the relationship of the full values of E_A and E_B. As shown in Table 1, prediction 706 may be one of four values: 0 (predicting $E_A=E_B$), +1 (predicting $E_A=E_B+1$), -1 (predicting $E_B=E_A+1$), and X (predicting $d \geq 1$, meaning the result of close path 240 is invalid). It is noted that in other embodiments, different values for prediction 706 are possible.

E_{A1}	E_{A0}	E_{B1}	E_{B0}	Pred.
0	0	0	0	0
0	0	0	0	-1
0	0	1	0	X
0	0	1	1	+1
0	1	0	0	+1
0	1	0	1	0
0	1	1	0	-1
0	1	1	1	X
1	0	0	0	X
1	0	0	1	+1
1	0	1	0	0
1	0	1	1	-1
1	1	0	0	-1
1	1	0	1	X
1	1	1	0	+1
1	1	1	1	0

Table 1

Because exponent prediction unit 704 only operates on the two least significant bits, the prediction may often be incorrect, due to differences in the upper order bits not considered by unit 704. For this reason, in

one embodiment, the actual exponent difference is computed in far data path 230 and utilized as a final select signal to determine whether far path 230 or close path 240 includes the correct result value.

Data path 240 further includes a shift-swap unit 710, which is coupled to receive an exponent prediction from unit 704, as well as mantissa values M_A and M_B from input unit 210. Shift-swap unit 710, in response to receiving the input mantissa values, generates shifted mantissa values 712A-B, which are conveyed to an adder unit 720. Unit 710 additionally generates a guard bit 714 which is conveyed to selection unit 730. Adder unit 720 is configured to generate a plurality of outputs (722A-B), representing sum and sum+1, respectively. Adder unit 720 also conveys a plurality of signals to selection unit 730 as will be described below. Selection unit 730, in response to receiving an exponent prediction from unit 704 and a plurality of control signals from adder unit 720 and shift-swap unit 710, generates a close path select signal 732, conveyed to a multiplexer-inverter unit 740. Signal 732 is usable to select either adder output 722A or 722B to be conveyed as close path preliminary result 742. Result 742 is conveyed to a left shift unit 750, which also receives a shift value from selection unit 730 and a predicted shift amount 772. Left shift unit 750 is configured to shift close path preliminary result 742 left by a number of bits indicated by shift amount 772. In one embodiment, the shift value conveyed by selection unit 730 is shifted in at the LSB.

The output of left shift unit 750 is the mantissa portion of close path result 242. The exponent portion of close path result 242 is generated by an exponent adjustment unit 780, which receives the largest input exponent value 309 from far data path 230. Unit 780 is configured to adjust exponent 309 by predicted shift amount 772 to produce the final close path exponent. As will be described below, the value of this exponent portion may be off by one in some cases due to the nature of the prediction mechanism. In one embodiment, this possible error is checked and corrected if needed in the final multiplexer stage.

Predicted shift amount 772 is the output of a shift prediction unit 752. Unit 752, in one embodiment, is coupled to receive three sets of inputs at prediction units 754A-C. Prediction unit 754A is coupled to receive an unshifted version of mantissa value M_A, and a negated version of M_B which is right-shifted by one bit (this represents a prediction that operand 204A has an exponent value one greater than the exponent value of operand 204B). Prediction unit 754B is coupled to receive unshifted, non-negated versions of M_A and M_B, representing a prediction that the exponent values of both operands are equal. Finally, prediction unit 754C is coupled to receive an unshifted version of mantissa value M_B and a negated version of M_A which is right-shifted by one bit (representing a prediction that operand 204B has an exponent value one greater than the exponent value of operand 204A). The predictions of units 754A-C are concurrently conveyed to a shift prediction multiplexer 760, which receives an exponent prediction from unit 704 as a control signal. The output of shift prediction multiplexer 760 is conveyed to a priority encoder 770, which generates predicted shift amount 772.

Turning now to Fig. 13, a block diagram of one embodiment of shift-swap unit 710 is shown. As shown, shift-swap unit 710 is coupled to receive exponent prediction value 706 from exponent prediction unit 704, as well as mantissa values M_A and M_B from input unit 210. Exponent prediction value 706 is conveyed to a pair of operand multiplexers 802A-B, as well as a guard bit generation unit 804.

Operand multiplexer 802A is coupled to receive unshifted versions of M_A and M_B, while operand multiplexer 802B receives an unshifted version of M_B and versions of M_A and M_B which are right shifted by one bit. These right shifted values are generated by a pair of right shift units 806. (In one embodiment, the shift

units 806 simply route the bits of the input values one place rightward, appending a "0" as the MSB). If exponent prediction value 706 indicates that $E_A=E_B$, operand multiplexer 802A selects M_A to be conveyed as shift output 712A and operand multiplexer 802B selects M_B to be conveyed as shift output 712B. The output of guard bit generation unit 804, G bit 714, is not used (in one embodiment) in the equal exponent case. If exponent prediction 706 indicates that $E_A=E_B+1$, operand multiplexer 802A selects M_A to be conveyed as shift output 712A, and operand multiplexer 802B selects a one-bit-right-shifted version of M_B to be conveyed as shift output 712B. Additionally, the bit shifted out of M_B is conveyed as guard bit 714. If exponent prediction 706 indicates that $E_B=E_A+1$, operand multiplexer 802A selects M_B to be conveyed as a shift output 712A, while operand multiplexer 802B selects a one-bit-right-shifted version of M_A to be conveyed as shift output 712B. Additionally, the bit shifted out of M_A is conveyed as guard bit 714. (If exponent prediction value 706 predicts the exponents are not valid close path values, the output of shift-swap unit 710 is undefined in one embodiment since the far path result is selected in such a case).

Since, in the embodiment shown, shift-swap unit 710 ensures that operand 712A is larger than operand 712B, the exponent difference for subsequent operations within close data path 240 is either 0 or 1 (-1 is no longer applicable). Accordingly, logic unit 810 is configured to receive exponent prediction value 706 and generate a corresponding exponent equality signal 812. As will be described below, exponent equality signal is utilized in selection unit 730 in order to generate close path select signal 732.

Because in the embodiment shown, close path 240 handles only subtraction operations, the output of multiplexer 802B, 712B, is inverted (one's complemented) before conveyance to adder unit 720.

Turning now to Fig. 14, a block diagram of one embodiment of adder unit 720 is depicted. As shown, adder unit 720 includes a pair of adders units, 900A-B. Adder unit 900A receives shift outputs/adder inputs 712A-B and carry in signal 904A, and generates an adder output 722A. Similarly, adder unit 900B receives shift outputs/adder inputs 712A-B and carry in signal 904B, and generates adder output 722B. Adder unit 720 generates outputs corresponding to sum and sum+1 by having carry in signal 904A at a logical zero and carry in signal 904B at a logical one.

As will be described below, selection unit 730 generates a signal which selects either adder output 722A or 722B based upon a number of input signals. Adder unit 720 conveys a number of signals to selection unit 730 which are used in this calculation. These signals include sign bits A_S and B_S of operands 204, most significant bits A_{MSB} and B_{MSB} of operands 204, carry in signal 906 to MSB adder block 902B, and least significant bit S_{LSB} of result 722A. As with adders 400 described with reference to Fig. 8 above, adders 900A-B may be implemented as a single adder producing sum and sum+1.

Turning now to Fig. 15, a block diagram of one embodiment of selection unit 730 is depicted. As shown, selection unit 730 receives a number of inputs in the embodiment shown, including least significant bit S_{LSB} (L) from adder unit 720, guard bit (G) 714 from shift-swap unit 710, most significant bit B_{MSB}, C_{MSB} 906, and exponent equality signal 812, indicating whether exponents E_A and E_B are equal or differ by one. Selection unit 730 includes a selection logic block 950, which includes a plurality of selection sub-blocks 950A-D. Each sub-block 950A-D generates a corresponding select signal 952. Selection sub-block 950D also generates a shift value 954, which is conveyed to left shift unit 750. Select signals 952A-D are conveyed to a close path

result multiplexer 960, which also receives a plurality of control signals. These control signals include exponent equality signal 812, an MSB value 956, and a sign value 958.

In one embodiment, MSB value 956 and sign value 958 are generated by a prediction select unit 962. As shown, prediction select unit 962 includes two multiplexers 970A-B. Multiplexer 970A is coupled to receive B_{MSB}, and also has another input hardwired to receive a logic high signal. The output of multiplexer 970A, C_S 957, is selected by C_{MSB} 906. C_S 957 is inverted by inverter 972 and conveyed as sign value 958, representing the sign of the output of adder unit 720. Multiplexer 970B, on the other hand, is configured to receive inverted and non-inverted versions of B_{MSB}. C_{MSB} also provides selection for multiplexer 970B. The output of 970B is conveyed to multiplexer 960 as the MSB of the output of adder unit 720.

Because close data path 240 performs subtraction operations for a limited set of operands ($E_{diff} \leq 1$), only a small number of cases must be considered in order to perform prediction of selection values. In the embodiment shown, there are four cases (corresponding to four predicted select values 952) covered by selection logic 950. Selection sub-block 950A corresponds to the case in which the operand exponents are equal (E_A=E_B) and the subtraction result is positive (M_A>M_B). For this particular case, since there is no borrow from the guard bit position, the output of selection sub-block 950A (952A) always indicates a predicted selection of adder output 722B (sum+1). Selection sub-block 950B corresponds to the case in which the operand exponents are equal (E_A=E_B) and the subtraction result is negative (M_A<M_B). Since this case results in a negative number, the output of selection sub-block 950B (952B) always indicates a predicted selection of adder output 722A (sum). (As will be described below, this value is later inverted to return it to sign-magnitude form). Selection sub-block 950C corresponds to the case in which the exponent values differ by one (E_A=E_B+1) and adder output 722A (sum) is not normalized (S_{MSB}=0). It is noted that, in the embodiment shown, at this stage in the pipeline, the possible exponent difference is either 0 or 1 since the operands are swapped (if needed) in shift-swap unit 710. Thus, while an exponent difference of -1 may exist for operands entering close data path 240, the inputs to selection logic block 950 have an exponent difference of either 0 or 1. Selection sub-block 950C generates a predicted selection value (952C) equal to the complement of guard bit 714. If the guard bit is zero, there is no borrow from the LSB, and adder output 722B (sum+1) is indicated by selection value 952C. Furthermore, shift value 954 is zero. Conversely, if the guard bit is one, there is a borrow from the LSB. This effectively cancels out the need for correction of one's complement subtraction, accordingly, adder output 722A (sum) is selected (and guard bit 714 is conveyed as shift value 954). Lastly, selection sub-block 950D corresponds to the case in which the exponent values differ by one (E_A=E_B+1) and adder output 722A (sum) is normalized (S_{MSB}=1). Selection sub-block 950D generates a predicted selection value (952D) which is indicative of (sum+1) according to the equation L+G', where G' represents the complement of guard bit 714. (If G=0, there is no borrow from the LSB and sum+1 is selected. If L=0 and G=1, there is a borrow, so sum is selected. If L=1 and G=1, there is a borrow, but rounding occurs, so sum+1 is selected).

It is noted that in one embodiment, selection logic 730 includes a separate zero detect unit which is configured to recognize the case when the result of the close path subtraction is zero (E_A=E_B and M_A=M_B). A separate zero detect unit may be utilized because in floating point representations such as IEEE standard 754, zero values are treated in a special fashion. A zero detect unit is not pictured in Fig. 15 for simplicity and clarity.

Select signals 952A-D are conveyed to close path result multiplexer 960. The control signals also received by multiplexer 960 are usable to convey one of select signals 952 as close path select signal 732. As described above, these control signals for multiplexer 960 include, in one embodiment, exponent equality value 812, MSB value 956, and sign value 958. Exponent equality signal 812 is usable to determine whether close path select signal is one of signals 952A-B (equal exponents) or 952C-D (unequal exponents). If exponent equality signal 812 is indicative of equal exponents, sign value 958 is usable to determine whether adder output 722A is positive or negative. Accordingly, either signal 952A or 952B may be selected. Alternately, if exponent equality signal 812 is indicative of unequal exponents, MSB value 956 may be utilized to determine whether adder output 722A is properly normalized, allowing for selection of either signal 952C or 952D.

Although sign and MSB values are generated by adder unit 720 and are included in adder output 722A, MSB value 956 and sign value 958 are generated in parallel by selection unit 730. This allows close path select signal to be determined more quickly and speed operation of close data path 240. In order to perform this parallel generation, B_{MSB} and C_{MSB} are conveyed from adder unit 900A. (It is noted that for the embodiment of close data path 240 depicted in Fig. 15, $A_{MSB}=1$, $A_S=1$, and $B_S=1$. This allows the logic of prediction unit 962 to be simplified).

MSB value 956 is generated by multiplexer 970B using C_{MSB} 906, which is the carry in signal to the MSB of adder output 722A. Because it is known that $A_{MSB}=1$, S_{MSB} is thus equal to B_{MSB}' if $C_{MSB}=0$, and B_{MSB} if $C_{MSB}=1$. MSB value 956 may thus be quickly generated and conveyed to multiplexer 960.

Sign value 958 is generated by multiplexer 970A and inverter 972. Because $A_{MSB}=1$ for close data path 240, a carry out of the MSB of adder output 722A (referred to in Fig. 15 as C_S) is dependent upon C_{MSB} 906. If C_{MSB} 906 is 0, C_S 957 is equal to B_{MSB}; otherwise, C_S 957 is 1. With $A_S=1$ and $B_S=0$, the sum of the sign bit of adder output 722A is thus equal to the inverted value of C_S 957. The output of inverter 972 is conveyed to multiplexer 960 as sign value 958.

Other embodiments of prediction selection unit 962 are also contemplated. For instance, C_{MSB} signal 957 may be directly conveyed from adder unit 900A instead of being generated by prediction selection unit 960. Various other embodiments of unit 960 are also possible.

Turning now to Fig. 16A, an example 1000A of subtraction within close data path 240 is shown according to one embodiment of the invention. Example 1000A is representative of the close path case predicted by selection sub-block 950A, in which $E_A=E_B$ and $M_A>M_B$. Because guard bit 714 is zero in this case, no borrowing is performed and the correction for one's complement addition is always needed. (This can be seen in the difference between actual result 1002A and computed result 1002B, which corresponds to adder output 722A). As a result, adder output 722B, or sum+1, is indicated by select signal 952A.

Turning now to Fig. 16B, an example 1000B of subtraction within close data path 240 is shown according to one embodiment of the invention. Example 1000B is representative of the close path case predicted by selection sub-block 950B, in which $E_A=E_B$ and $M_B>M_A$. As with example 1000A, guard bit 714 is zero in this case, so borrowing is not performed. Because M_B is larger than M_A, however, the subtraction result is negative. It is noted that actual result 1004A is the bit-inverted (one's complement) of computed result 1004B, which corresponds to adder output 722A. Accordingly, actual result 1004A may be computed by

selecting adder output 722A for this case, inverting the resultant mantissa, and setting the sign bit of the result to indicate a negative number. This relationship may be seen from the following formulas:

$$S = A + B'; (4)$$

$$S = A + 1\text{'s comp}(B); (5)$$

$$S' = 1\text{'s comp}(A + 1\text{'s comp}(B)); (6)$$

$$S' = 2^N - (A+2^N-B-1) - 1; (7)$$

$$S' = B - A. (8)$$

Turning now to Fig. 16C, an example 1000C of subtraction within close data path 240 is shown according to one embodiment of the invention. Example 1000C is representative of the close path case predicted by selection sub-block 950C, in which $E_A=E_B+1$ and $S_{MSB}=0$. As shown in Fig. 15, adder output 722B (sum+1) is indicated by select signal 952C according to the equation G'. As can be seen in example 1000C, the fact that G=0 results in no borrowing, and actual result 1006A is equal to computed result 1006B plus one. Accordingly, adder output 722B (sum+1) is selected.

Turning now to Fig. 16D, an example 1000D of subtraction within close path 240 is shown for the case predicted by selection sub-block 950C in which G=1. In this case, there is a borrow from the LSB since guard bit 714 is set. Accordingly, select signal 952C is indicative of adder output 722A (sum). This can be seen from the fact that actual subtraction result 1008A is equal to computed subtraction result 1008B.

Turning now to Fig. 16E, an example 1000E of subtraction within close path 240 is shown for the case predicted by selection sub-block 950D in which L=0 and G=1. Example 1000E is representative of the close path case predicted by selection sub-block 950D, in which $E_A=E_B+1$ and $S_{MSB}=1$. As shown in Fig. 15, adder output 722B (sum+1) is indicated by select signal 952D according to the equation L+G'. In example 1000E, a borrow is performed, canceling out the need for the one's complement correction. Furthermore, no rounding is performed since L=0. Accordingly, adder output 722A (sum) is selected by select signal 952D. This can be seen from the fact that actual subtraction result 1010A in Fig. 16E is equal to computed subtraction result 1010B.

Turning now to Fig. 16F, an example 1000F of subtraction within close path 240 is shown for the case predicted by selection sub-block 950D in which L=1 and G=0. In contrast to example 1000E, no borrow is performed in example 1000F, necessitating a one's complement correction of +1. Accordingly, adder output 722B (sum+1) is selected by select signal 952D. This can be seen from the fact that actual subtraction result 1010A in Fig. 16E is equal to computed subtraction result 1010B plus one.

Turning now to Fig. 16G, an example 1000G of subtraction within close path 240 is shown for the case predicted by selection sub-block 950D in which L=1 and G=1. As with example 1000E, a borrow is performed from the LSB, cancelling the need for a one's complement correction of +1. Because both the LSB and guard bit are set in the result, however, the subtraction result is rounded up, according to an embodiment in which results are rounded to the nearest number (an even number in the case of a tie). Accordingly, even though actual subtraction result 1014A and computed subtraction result 1014B are equal, adder output 722B is selected,

effectively rounding the difference value to the nearest number (which is chosen to be the even number since the computed subtraction result 1014B is halfway between two representable numbers).

Turning now to Fig. 17, a block diagram of one embodiment of multiplexer-inverter unit 740 is shown. Unit 740 is configured to select one of adder outputs 722 as close path preliminary result 742. Result 7412 is then conveyed to left shifter 750, described below with reference to Fig. 18.

Multiplexer-inverter unit includes an AND gate 1106, a bit XOR block 1110, and a close path result multiplexer 1100. Bit XOR block 1110 is coupled to receive adder output 722A, as well as XOR enable signal 1108 from AND gate 1106. XOR enable signal 1108 is asserted for the case (described above with reference to Fig. 16B) in which E_A=E_B and M_B>M_A. Bit XOR block 1110, in one embodiment, includes a two-input XOR gate for each bit in adder output 722A. One input of each XOR gate is a corresponding bit of output 722A; the other bit is XOR enable signal 1108. If signal 1108 is de-asserted, then, XOR block output 1104 is identical to adder output 722A. If signal 1108 is asserted, however, XOR block output 1104 is equal to the one's complement of adder output 722A. Signal 1108 is only enabled for the case in which the result of the close path subtraction is negative.

In addition to receiving XOR block output 1104, close path result multiplexer 1100 also receives adder output 722B. Close path select signal 732, calculated in selection unit 730 as described above, is usable to select either output 1104 or 722B to be conveyed as close path preliminary result 742. Result 742 is then conveyed to left shift unit 750, described next with reference to Fig. 18.

By selecting sum or sum+1 as preliminary result 742, multiplexer-inverter unit 740 is configured to quickly perform the IEEE round-to-nearest operation. By generating more than one close path result and selecting from between the results (according to various rounding equations), a result 742 is generated for forwarding to a normalization unit (left shifter). The value conveyed to the normalization unit of Fig. 18 is such that shifted output value is correctly rounded to the nearest number. This rounding apparatus advantageously eliminates the need to perform an add operation (subsequent to the add operation of adder unit 720) in order to perform rounding. Additionally, recomplementation is also achieved quickly since adder output 722A need only be inverted rather than having to perform a two's complement invert and add.

Turning to Fig. 18, a block diagram of one embodiment of left shifter unit 750 is shown. As depicted, left shift unit 750 includes a left shift register 1200 and a shift control unit 1210. Shift control unit 1210 receives predicted shift amount 772 from shift prediction unit 752 and shift value 954 from selection logic 950C. In response to these inputs, shift control unit 1210 controls the number of bits the value in register 1200 is shifted leftward. Shift control unit 1210 additionally controls what bit is shifted in at the LSB of register 1200 with each left shift. The result after shifting is conveyed as close path result 242.

For close path subtraction operations, preliminary result 742 is either normalized or requires one or more bits of left shift for normalization. Furthermore, since the loss of precision due to operand alignment is at most one bit, only one value need be generated to shift in at the LSB. This value (shift value 954 in the embodiment shown) is shifted in at the LSB for the first left shift (if needed). If more than a one bit left shift is required, zeroes are subsequently shifted in at the LSB. The output of register 1200 is conveyed as close path result 242.

Turning now to Fig. 19, a block diagram of one embodiment of result multiplexer unit 250 is shown. As depicted, result multiplexer unit 250 includes a final result shift control unit 1310, a 1-bit left shift unit 1312, a exponent correction adder 1313, and a pair of final multiplexers 1320. Final multiplexer 1320A selects to the exponent portion of result value 252, while final multiplexer 1320B selects the corresponding mantissa portion. Final multiplexer 1320A receives the exponent portions of both far path result 232 and close path result 242. Additionally, multiplexer 1320A receives the output of adder 1313, equal to the close path exponent plus one. As will be described below, in some cases predicted shift amount 772 is one less than the shift value needed to normalize the mantissa portion of close path 242. If this is the case, the close path exponent is one less than its true value. Accordingly, in addition the far and close path exponent values, the output of adder 1313 is also conveyed to multiplexer 1320A. Similarly, multiplexer 1320B receives far and close mantissa portions, along with a corrected close path mantissa value generated by shift unit 1312. The corrected close path mantissa value is generated for the case in which the mantissa of close path result 242 is not properly normalized. Guard bit 714 is shifted into the LSB in such a case.

Shift control unit 1310 utilized exponent difference select 313 and close path MSB 1314 in order to generate final select signals 1322A-B. As described above, the actual exponent difference (calculated in far path 230) indicates whether far path result 232 or close path result 242 is to be selected. Exponent difference select 313 is thus used (along with signal 1314) to select one of the inputs to each of multiplexers 1320. If signal 313 indicates that the exponent difference is greater than one, far path result 232 exponent and mantissa portions are selected as result value 252. On the other hand, if the absolute exponent difference is indicated to be 0 or 1, close path MSB 1314 selects whether the calculated or corrected versions of close path result 242 are conveyed as result value 252.

As described above, predicted shift amount 772 is generated by a shift prediction unit 752. In one embodiment of close path 240, shift prediction unit 752 includes three leading 0/1 prediction units 754. Prediction unit 754A is for the case in which $E_A=E_B+1$, unit 754B is for the case in which $E_A=E_B$, and unit 754C is for the case in which $E_B=E_A+1$. As will be described below, units 754A and 754C may be configured to provide improved speed and reduced space requirements.

Turning now to Fig. 20, a block diagram of a prior art leading 0/1 prediction unit 1400 is depicted. Prediction unit 1400 is configured to receive two operands and generate an indication of the location of the leading 0 (or 1) in the result value. As will be described below, the prediction generated by unit 1400 is accurate to within one bit position. The operation of prediction unit 1400 is described in order to provide a contrast to an improved leading 1 prediction unit described below with reference to Fig. 26.

As shown, prediction unit 1400 includes a pair of operand input registers 1404A-B. Operand register 1404A receives operand A, storing bits A'_{MSB} to A'_{LSB}. Operand register 1404B receives a bit-inverted version of operand A, storing bits B'_{MSB} to B'_{LSB}. The contents of register 1404A are denoted as A' (even though $A'_i = A_i$) for purposes of consistency since the inverted contents of register 1404B are denoted as B'. Prediction unit 1400 further includes a TGZ logic stage 1408, which includes TGZ generation units 1410A-1410Z. (The TGZ generation unit which is coupled to A'_{LSB} and B'_{LSB} is denoted as "1410Z" simply to show that this unit is the final sub-block with logic stage 1408. The number of TGZ generation units 1410 within logic stage 1408 corresponds to the length of operands A and B). Each TGZ generation unit 1410 receives a pair of

corresponding bits from operands A and B and produces, in turn, outputs T, G, and Z on a corresponding TGZ bus 1412. TGZ generation unit 1410A, for example, produces T, G, and Z outputs on TGZ bus 1412A. Prediction unit 1400 further includes a leading 0/1 detection logic block 1418, which includes a plurality of sub-blocks 1420A-1420Z. Logic block 1418 typically includes either n or n+1 sub-blocks, where n is the number of bits in each of operands 1404. Each sub-block 1420 receives three TGZ bus 1412 inputs. Within prediction unit 1400, a given logic sub-block 1420 has a corresponding TGZ generation unit 1410. TGZ generation unit 1410B, for example, corresponds to logic sub-block 1420B. Generally speaking, then, a given logic sub-block 1420 receives TGZ bus values from its corresponding TGZ generation unit, from the TGZ generation unit corresponding to the next most significant sub-block 1420, and from the TGZ generation unit corresponding to the next least significant sub-block 1420. (As shown, logic sub-block 1420B receives TGZ bus 1412B from unit 1410B, TGZ bus 1412A from unit 1410A, and TGZ bus 1412C from unit 1410C. Unit 1410C is not pictured in Fig. 20). The first and last sub-blocks 1420 receive predefined TGZ values in one embodiment in order to handle the boundary cases. Each logic sub-block 1420 generates a prediction bit value 1430. Each value 1430 is usable to indicate the presence of leading 0 or 1 bits in its corresponding bit position. Collectively, values 1430A-Z make up leading 0/1 detection bus 1428. As will be described below, prediction unit 1400 may be optimized to reduce space requirements and increase performance. Such an improved prediction unit is described below with reference to Fig. 26. This prediction unit is particularly useful for speeding leading 1 predictions performed in close path 240 of add/subtract pipeline 220.

Turning now to Fig. 21, a logic diagram of prior art TGZ generation unit 1410 is depicted. Unit 1410 shown in Fig. 21 is representative of units 1410A-Z shown in Fig. 20. As shown, unit 1410 includes logic gates 1502A, 1502B, and 1502C, each of which receives inputs A'_i and B'_i, where i indicates a corresponding bit position within A and B. In one embodiment, logic gate 1502A is an AND gate which generates an asserted value G_i when both A'_i and B'_i are both true. Logic gate 1502B is an exclusive-OR gate which generates an asserted T_i value if one of A'_i and B'_i is true. Finally, logic gate 1502C is a NOR gate which generates an asserted Z_i value if A'_i and B'_i are both zero. The values G_i, T_i, and Z_i make up TGZ bus 1412 for bit position i.

For the configuration of logic gates shown in Fig. 21, one (and only one) of signals T, G, and Z is asserted for each bit position in the result of A'+B'. Thus, for a given set of operands, the output of logic stage 1408 may be represented by a string of T's, G's, and Z's. It is known that a leading 1 may be predicted by matching the string T^*GZ^*, where the "*" may be read as "0 or more occurrences of". Conversely, a leading 0 may be predicted by matching the string T^*ZG^*. As stated above, predictions generated by using these strings may be subject to a 1-bit correction.

Turning now to Figs. 22A-C, examples of leading 0/1 prediction using T-G-Z strings are shown. Fig. 22A depicts an example 1600A of leading 1 prediction for the case of A-B, where A=10110b and B=10010b. As shown, the actual leading 1 position is found in the third most significant bit position of the subtraction result. This operation is performed in hardware as A'+B', where A' is equal to A and B' is the inverted version of B. For this set of input operands, the resulting T-G-Z string is shown as TTGTT. This string stops matching the regular expression T^*GZ^* in the fourth most significant bit position. The leading 1 is thus indicated as being in the last bit position which matches the target string (the third most significant bit), which happens for this case to be the correct prediction.

Turning now to Fig. 22B, another example of leading 1 prediction is shown. Example 1600B depicts the case of A-B, where A=10110b and B=10011b. For these operands; the actual leading 1 position is in the fourth most significant bit. When the subtraction is performed in hardware as A'+B', the resulting T-G-Z string is TTGTZ. As with example 1600A, this string stops matching in the third most significant bit. This results in a leading 1 prediction which is off by one bit position. In one embodiment, final result multiplexer 250 may be configured to correct this one-bit position error as described above.

Turning now to Fig. 22C, an example of leading 0 prediction is shown. Example 1600C depicts the case of A-B, where A=10010b and B=11001b. For this set of operands, the leading 0 is found in the third most significant bit position. When this subtraction is performed in hardware as A'+B', the resulting T-G-Z string is TZTGZ. This string stops matching the target string T^*ZG^* after the second bit position. This results in a leading 0 prediction which is off by one bit position.

Turning now to Fig. 23, a logic diagram is shown for leading 0/1 detection sub-block 1420 (representative of sub-blocks 1420A-Z in Fig. 20). As shown, sub-block 1420 includes logic gate 1702A-C, 1704A-C, 1706, 1708, and 1710. An asserted prediction bit value 1430 indicates that either a leading 0 or leading 1 is present in this bit position.

In one embodiment, when a leading 1 value is predicted, the output of one of AND gates 1702 is asserted. Each of AND gates 1702 receives values from the current bit position, the previous bit position, and the next bit position. An assertion of one of gates 1702 indicates that the T-G-Z string produced by logic stage 1408 stops matching the target string T^*GZ^* in the next bit position. Each logic sub-block 1420 includes these gates 1702 in order to correspond to each of the possible ways a string match may end. It is noted that only one of the outputs of AND gates 1702 may be asserted at a given time. An assertion of one of the outputs of gates 1702 causes the output of gate 1706, leading 1 prediction 1707, to also be asserted.

Conversely, AND gates 1704A-C correspond to leading 0 detection in one embodiment. Each of these gates also receives TGZ values from the current bit position, the previous bit position, and the next bit position. An assertion of one of gates 1704 indicates that the T-G-Z string produced by logic stage 1408 stops matching the target string T^*ZG^* in the next bit position. Each of sub-blocks 1420 includes three gates in order to correspond to each of the possible ways a string match may end. It is noted that only one of the outputs of AND gates 1704 may be asserted at a given time. An assertion of any of the outputs of gates 1704 causes the output of OR gate 1708, leading 0 prediction 1709, to also be asserted. OR gate 1710 asserts signal 1430 if either of signals 1707 or 1709 is asserted. The most significant position within result bus 1430A-Z which is asserted indicates the position of the leading 0 or 1.

The configuration of sub-block 1420 is typically used when both leading 0 and 1 determination is to be performed. As such, this configuration is used in prediction unit 754B. Prediction unit 754B corresponds to the indeterminate case in which $E_A=E_B$, and it is not known whether the subtraction operation A-B will produce a positive or negative result (leading 1 and leading 0 determination, respectively). As will be shown with reference to Fig. 24, prediction unit 1400 may be configured differently if more information is known regarding operands A and B.

Turning now to Fig. 24, a logic diagram of a prior art prediction unit sub-block 1800 is shown. Sub-block 1800 is another embodiment of logic sub-block 1420 shown in Fig. 20. Sub-block 1800 is usable for

operands with the restriction A>B. Sub-block 1800 receives T and Z values for each bit position in the sum of A'+B'. The T and Z values are coupled to inverters 1802A and 1802B, respectively. The outputs of inverters 1802, $\overline{T}_i$ and $\overline{Z}_i$, are coupled to an AND gate 1810, which conveys result bus 1820 as an output.

Sub-block 1800 illustrates an improved method for generating leading 1 prediction when A>B. (Leading 0 prediction is not relevant since the result of subtraction is positive for A>B). The configuration of sub-block 1800 is accomplished noting that the leading 1 target string T^*GZ^* stops matching when the current bit position is not a T and the next bit position is not a Z. A prediction unit which includes sub-block 1800 for each bit may omit logic for generating G on a bit-by-bit basis, since this signal is not utilized in order to generate result bus 1820. Although logic sub-block 1800 provides improved performance over logic sub-block 1420, the operation of a prediction unit may be further improved for the case of $E_A=E_B+1$, which is particularly important for the operation of close data path 240.

Turning now to Fig. 25, an illustration 1900 is shown depicting the derivation of an improved prediction unit 754A/C for close data path 240. As described above, operands in close data path 240 have an exponent difference E_{diff} of either 0, +1, or -1. Prediction unit 754B handles the $E_{diff}=0$ case, while units 754 and 754C handle the +1 and -1 cases, respectively. The example shown in illustration 1900 corresponds to the case in which $E_A=E_B+1$ (unit 754A), although it is equally applicable to the case in which $E_B=E_A+1$ (unit 754C) with a few minor modifications.

Illustration 1900 depicts operands A and B after operand B (the smaller operand in this case) is aligned with operand A. Because operand A is the larger operand, the MSB of A is a 1. Furthermore, since it is predicted that $E_A=E_B+1$, the MSB of B (after alignment) is a 0. Accordingly, the MSB of B' (the inverted version of B) is a 1. This combination of bits in the MSB results in a G value for the T-G-Z string corresponding to the result of A'+B'. The T-G-Z value of the subsequent bits in the result of A'+B' is not known. It may be ascertained however, that the next bit position which equals $\overline{Z}$ indicates that the target string T^*ZG^* stopped matching in the previous bit position. A prediction unit 754 which utilizes this detection technique is described with reference to Fig. 26.

Turning now to Fig. 26, a block diagram of one embodiment of prediction unit 754A/C is shown. As described above, unit 754A/C is optimized for the case in which $E_A=E_B+1$ (or $E_B=E_A+1$). Accordingly, the prediction unit shown in Fig. 26 is indicated as corresponding to unit 754A or 754C as shown in Fig. 12. Unit 754A/C includes input registers 2000A-B. Input register 2000A receives operand A, storing bits A'_{MSB} through A'_{LSB}, while input register 2000B receives a bit-inverted version of operand B, storing bits B'_{MSB} through B'_{LSB}. Prediction unit 754A/C further includes a plurality of OR gates 2002A-Z, each coupled to receive a pair of input values from input registers 2000. The outputs of OR gates 2002 are conveyed to output register 2010. The collective output of register 2010 (prediction bit values 2011A-Z) forms prediction string 2012. In one embodiment, prediction bit value 2011Z is hardwired to a logic high value in order to produce a default leading 1 value.

The prediction string 2012 generated by unit 754A/C is conveyed to shift prediction multiplexer 760. Multiplexer 760 receives prediction strings from each of prediction units 754, and is configured to choose a prediction string based on exponent prediction value 706. For example, if exponent prediction value 706

indicates that $E_A=E_B$, the prediction string conveyed by prediction unit 754B is selected by multiplexer 760. This string is then conveyed to priority encoder 770, which converts the string into predicted shift amount 772.

As described above, given the restriction that $E_A=E_B+1$, the contents of output register 2010 may be performed by using a single OR gate for each bit position. As shown in Fig. 25, the first T-G-Z value of the result A'+B' is a G. (This results from A having an MSB of 1 and the inverted version of B, B', also having an MSB of 1). Given a starting string value of G, the result stops matching the target string of T^*GZ^* when $\overline{Z}$ is encountered in a bit position. Therefore, when the first $\overline{Z}$ value is detected at a particular bit position i, the prediction bit value 2011 for bit position i+1 (where i+1 is one bit more significant than position i) should indicate that a leading one value is present.

Such a configuration is shown in Fig. 26. Prediction bit value 2011A is asserted if either the second most significant bit of A' or the most significant bit of B' is set. (It is noted that the bit values conveyed to OR gates 2002 from operand B' have a 1-bit relative bit position to those bit values conveyed from operand A'. This routing effectively performs the functionality of aligning A' and B'. In another embodiment, B' may be shifted prior to conveyance to register 2000B. In such a case, the bit values routed to a particular gate 2002 would have common relative bit positions within input registers 2000). If either of these bits is set the second T-G-Z value in the result string is either G or T, but not Z. Accordingly, the strings stops matching in the second most significant bit position. This corresponds to a leading one being present in the most significant bit position. Hence, prediction bit value 2011A is asserted. The remaining prediction bit values 2011 are formed similarly. The final prediction bit value 2011Z is hardwired to a logical one (as a default in case none of the other bits are set). It is noted that although many bit values within prediction string 2012 may be asserted, typically only the most significant asserted position is utilized in determining the leading 1 position.

Prediction unit 754A/C achieves an optimal implementation of leading 1 prediction for the case in which $E_A-E_B=\pm1$. This case is particularly useful in close data path 240. Prediction unit 754A/C represents a considerable space savings relative to designs such as that shown in Fig. 24. For Fig. 24, each bit position includes an XOR gate (to generate T_i), a NOR gate (to generate Z_i), two inverters, and a final AND gate. Prediction unit 754A/C includes just a single OR gate for each bit position. Furthermore, each value within prediction string 2010 is generated using bit values from only a single bit position (two bits) in the input operands. This is in contrast to prior art designs in which prediction values are generated using bit values from at least two positions (for a total of four input bits). Such a prediction unit may provide considerable space savings (up to 75% relative to prior art designs). The speed of such a prediction unit is also correspondingly increased due to fewer gate delays.

As described above, the use of far data path 230 and close data path 240 provides an efficient implementation of add/subtract pipeline 220 by eliminating operations not needed for each path. The versatility of add/subtract pipeline 220 may also be increased by expanding the pipeline to handle additional operations. Figs. 27-30 describe an embodiment of far data path 230 which is configured to perform floating point-to-integer conversions. Similarly, Figs. 31-99 describe an embodiment of close data path 240 which is configured to perform integer-to-floating point conversions. As will be shown below, this additional functionality may be achieved with only a minimal number of hardware changes.

Turning now to Fig. 27A, a floating point number 2100 is shown (in single-precision IEEE format) along with its corresponding integer equivalent, integer number 2102. As shown, number 2100 is equal to 1.00111010011110100001101 x 2^{16}. (The exponent field in number 2100 includes a bias value of +128). Integer number 2102 represents the integer equivalent of floating point number 2102, assuming a 32-bit integer format (with one bit designated as the sign bit). Accordingly, to convert floating point number 2100 to its integer equivalent, the floating point mantissa is shifted such that the most significant bit of the mantissa (in one embodiment, a leading "1" bit) ends up in the bit position representing the floating point exponent (16) in the integer format. As shown, depending on the value of the floating point exponent, not all bits of the floating point mantissa portion may be included in the integer representation.

Turning now to Fig. 27B, a floating point number 2200 is shown along with corresponding integer representation, integer number 2202. As shown, number 2200 is equal to -1.1 x 2^{30}, with an implied leading "1" bit. Because the true exponent of floating point number 2200 (30) is greater than the number of mantissa bits (23+hidden 1), integer number 2202 includes all mantissa bits of the original number.

Turning now to Fig. 28, a block diagram of one embodiment of far data path 2300 is shown. Far data path 2300 is similar to far data path 230 described above with reference to Fig. 6; however, far data path 2300 is modified in order to perform floating point-to-integer (f2i) conversions. The components of far data path 2300 are numbered similarly to the components of far data path 230 in order to denote similar functionality.

Exponent difference unit 2310A receives exponent values E_B and E_A as in far data path 230. Exponent difference unit 2310B, however, receives the output of a multiplexer 2302 and exponent value E_B, where E_B corresponds to the floating point value which is to be converted to integer format. Multiplexer 2302 receives an exponent value E_A and a maximum integer exponent constant, and selects between these two values based on an f2i signal 2304. In one embodiment, signal 2304 is generated from the opcode of an float-to-integer conversion instruction. In the case of standard far path addition/subtraction, f2i signal 2304 is inactive, and E_A is conveyed to exponent difference unit 2310B. If signal 2304 is active, however, this indicates that a floating point-to-integer conversion is being performed on the floating point number represented by E_B and M_B. In this case, multiplexer 2302 conveys the maximum integer exponent constant to exponent difference unit 2310B.

The maximum integer exponent is indicative of the exponent of largest possible floating point value which may be converted to an integer (without clamping) by far data path 2300. If far data path 2300 is configured to handle the 32-bit signed integer format shown in Figs. 27A-B, the value 31 is used as the maximum integer exponent constant. In one embodiment, far data path 2300 may be configured to convert floating point numbers to different size integer formats. In such a case, a plurality of maximum exponent values may be multiplexed (selected by a size select signal) to provide the second input to multiplexer 2302.

For standard addition/subtraction in far data path 2300, exponent difference units 2310A-B operate as described above. For f2i conversions, however, only the shift amount 2312B generated by unit 2310B is utilized. As will described below, shift amount 2312A is effectively discarded since the "A" operand is set to zero in one embodiment of the f2i instruction. Shift amount 2312B, on the other hand, represents the amount that M_B has to be shifted in order to provide the proper integer representation. For a floating point input of 1.0 x 2^{30}, shift amount 2312B would be computed as 31-30=1.

To allow far data path 2300 to accommodate f2i conversions, the entire data path is configured to handle max(m, n) bits, where m is the number of bits in mantissa values M_A and M_B, and n is the number of bits in the target integer format. In other words, far data path 2300 is wide enough to handle the largest possible data type for its defined operations. In order to perform f2i conversion for 32-bit integers, then, right shift units 314 are 32 bits wide. Shift units 314A-B receive mantissa values M_A and M_B, respectively, each of which is left aligned. Shift outputs 2316A-B are then conveyed to multiplexer-inverter unit 2330.

Multiplexer-inverter unit 2330 receives shift outputs 2316, along with M_A, M_B, and an operand which is set to zero. (It is also noted that in another embodiment, mantissa value M_A may itself be set to zero before conveyance to far data path 2300). Unit 2330, in response to receiving f2i signal 2304, is configured to convey the zero operand as adder input 2332A and the shifted version of M_B as adder input 2332B. By setting add/subtract indication 202 to specify addition for the f2i conversion function, adder output 2342A is equal to adder input 2332B (M_B). Selection unit 2350 is thus configured to select adder output 2342A (sum) to perform the f2i operation.

Adder unit 2340, as described above, produces sum and sum+1 outputs in response to the adder inputs. For f2i conversions, however, since one operand is zero, adder output 2342A is equal to adder input 2332B. Accordingly, selection unit 2350, in response to receiving f2i signal 2232, selects adder output 2342A (sum) within multiplexer-shift unit 2360.

A multiplexer 2306 coupled between exponent adjust unit 2370 and multiplexer-shift unit 2360 is configured to provide the proper upper order bits for one embodiment of far path result 232. For standard far path operation (add and subtract operations), 24 bits (in one embodiment) of mantissa value are conveyed as the 24 least significant bits of result 232. Sign and exponent portions are conveyed as the upper order bits. Hence, when f2i signal 2304 is inactive, the output of exponent adjust unit 2370 and a sign bit (not shown) is conveyed as the upper order bits of far path result 232. On the other hand, when signal 2304 is active, the upper order bits of adder output 2342A are conveyed as the upper order bits of far path result 232. For one embodiment of f2i conversions, far path result 232 includes one sign bit followed by 31 integer bits. As will be described below, floating point values above or below the maximum/minimum integer values are clamped to predetermined values. In one embodiment of a 32-bit representation, these maximum and minimum integer values are 2^{31}-1 and -2^{31}, respectively.

Turning now to Fig. 29, a block diagram of one embodiment of multiplexer-inverter unit 2330 is depicted. Unit 2330 is modified slightly from multiplexer-inverter unit 330 described above with reference to Fig. 7 in order to handle floating point-to-integer conversions.

As shown, multiplexer-inverter unit 2330 includes control unit 2431, input multiplexers 2434A-B, and inverter 2436. Input multiplexer 2434A receives three inputs: M_A, M_B, and an zero operand set to zero, while input multiplexer 2434B receives the outputs 2316A-B of shift units 2314. Multiplexer 2434B receives another version of shift output 2316B as described below.

During standard operation of far data path 2300, two 24-bit floating point mantissas are added by adder unit 2340. In order to accommodate 32-bit integer values, however, adder unit 2340 (and other elements of data path 2300) are 32 bits wide. Accordingly, the 24-bit M_A and M_B values are routed to the least significant 24 bits of the adder (with the upper order bits padded with zeroes) in order to perform addition and subtraction. For the

case in which $E_A>E_B$, control unit 2431 generates select signals 2433 such that multiplexer 2434A selects M_A and multiplexer 2434B selects the 24-bit version of M_B (shift output 316B). Conversely, for the case in which $E_B>E_A$, select signals 2433 are generated such that multiplexer 2434A selects M_B and multiplexer 2434B selects the 24-bit version of M_A (shift output 2316A).

In one embodiment, far data path 2300 performs the f2i function by adding zero to an appropriately shifted version of operand B, using the sum as the integer result. If f2i signal 2304 is active, control unit 2431 generates select signals 2433A-B so that the zero operand is selected by multiplexer 2434A as adder input 2332A and that the 32-bit version of shift output 2316B is selected by multiplexer 2434B. For the f2i instruction/function, inverter 2436 is inactive in one embodiment. Hence, the output of multiplexer 2434B is conveyed as adder input 2332B.

For floating point-to-integer conversions, the exponent value of the floating point number may often exceed the maximum representable integer value. In one embodiment, if an overflow (or underflow) occurs, the converted integer may be clamped at the maximum (or minimum) representable value to provide a usable result for subsequent operations. An example of result clamping for the f2i instruction is described below with reference to Fig. 30.

Turning now to Fig. 30, a block diagram of one embodiment of result multiplexer unit 2500 is depicted. Unit 2500 is similar to multiplexer unit 250 depicted in Fig. 19, with additional hardware added to perform clamping of f2i conversion results. As shown, result multiplexer unit 2500 includes comparators 2504A-B, a shift control unit 2510, a left shift unit 2512, and a final multiplexer 2520.

Like final multiplexer 1320, multiplexer 2520 is configured to select result value 252 from a plurality of inputs according to a final select signal 2522 generated by shift control unit 2510. Control unit 2510 generates select signal 2522 from exponent difference select 2313, comparator outputs 2504A-B, and the most significant bit of close path result 242 (denoted in Fig. 30 as numeral 2514). Exponent difference signal 2313 is indicative of either far path result 232 or close path result 242, with an additional indication of whether far path result 232 is an f2i result. If signal 2313 does indicate that far path result is an f2i result, comparator outputs 2506 indicate whether the f2i result should be clamped. Comparator 2504A indicates an overflow if E_B (the original floating point exponent of operand B) is greater than or equal to 31, since the maximum positive integer for the embodiment shown is 2^{31}-1. Similarly, comparator 2504B indicates an underflow if E_B is greater than 31 or E_B=31 and M_B is greater than 1.0. If exponent difference select signal 2313 is indicative of close path result 242, either result 242 or its one-bit left shifted version (the output of shifter 2512) is chosen, depending on the whether result 242 is properly normalized.

As described above, far data path 2300 is similar to far data path 230, but with the additional f2i functionality. Because minimal hardware is needed to handle this extra instruction, the versatility of data path 2300 is increased with relativity little overhead. This provides an effective implementation of f2i conversion instructions through re-use of existing hardware. Similarly, integer-to-floating point conversion (i2f) may also be performed within add/subtract pipeline 220. One embodiment of pipeline 220 is described below with reference to Figs. 31-35 in which i2f conversions are performed in close data path 240.

Turning now to Fig. 31A, a 32-bit integer number 2550 is shown along with its corresponding IEEE single-precision equivalent 2552. The quantity represented by both numbers is 1.1 x 2^{30}. Because the number

of significant bits (2) in number 2550 is less than the number of mantissa bits in number 2552, no precision is lost. It is noted that in the embodiment shown, the mantissa portion of floating point number 2552 has a hidden 1 bit.

Turning now to Fig. 31B, a 32-bit integer number 2560 is shown along with its corresponding single-precision IEEE floating point equivalent 2562. Unlike integer 2550, integer 2560 includes more significant bits than are available in the mantissa portion of floating point number 2562. Accordingly, these extra bits are lost in the conversion process. It is noted that if the target floating point format includes a larger number of bits than are in the source integer format, no precision is lost during integer-to-float conversions.

Turning now to Fig. 32, a block diagram of one embodiment of close data path 2600 is depicted. Close data path 2600 has a similar structure to that of close data path 240 described above with reference to Fig. 12, but data path 2600 is additionally configured to perform i2f conversions. The differences in functionality between data path 240 and data path 2600 are described below. Other embodiments are possible in which the leading 1 bit is explicit.

In one embodiment, i2f conversions are performed by setting operand A to zero. Accordingly, multiplexer 2601 receives both mantissa value M_A and an operand set to zero. An i2f signal 2602 is utilized to select one of these input values to be conveyed as the output of multiplexer 2601. If i2f signal 2602 is inactive, mantissa value M_A is conveyed to both shift-swap unit 2610 and prediction 2654B, in which case close data path 2600 operates identically to close data path 240. If i2f signal 2602 is active, however, the zero operand is conveyed to both units 2610 and 2654B. Shift-swap unit 2610, in response to receiving i2f signal 2602, selects 0 and M_B to be conveyed as adder inputs 2620. In one embodiment, close data path 2600 is only configured to perform subtraction. In such an embodiment, a positive integer input to close data path 2600 produces a negative result from adder unit 2620 (since the integer is effectively subtracted from zero). In this case, as with close data path 240, the "sum" output of adder 2620 may be inverted in order to produce the correct result. Conversely, a negative integer input (in 2's complement form) to close data path 2600 produces a positive result from adder unit 2620. As will be described below, the 2's complement integer input is negated in shift-swap unit 2610 by taking the 1's complement. This results in an adder input having a magnitude which is one less than the original negative number. Accordingly, the correct output of adder unit 2620 is obtained by selecting the "sum+1" output, which corrects for the one's complement addition.

Restating, selection unit 2630 selects the output of adder unit 2620 based on the sign of operand B if i2f signal 2602 is active. If an i2f instruction is being performed, adder output 2622A (sum) is chosen (and subsequently inverted) if the sign of operand B is 0 (indicating a positive number). On the other hand, adder output 2622B (sum+1) is chosen if the sign of operand B is 1 (indicating a negative number). Multiplexer-inverter unit 2640, in response to receiving close path select signal 2632, conveys the selected adder output 2622 as close path preliminary result 2642.

Close path preliminary result 2642 is then normalized in left shift unit 2650 according to predicted shift amount 2672. If i2f signal 2602 is active, prediction unit 2654B receives a zero operand and a negated version of M_B as inputs. The prediction string generated by unit 2654B is then selected by shift prediction multiplexer 2660 in response to signal 2602. Priority encoder 2670 then generates a predicted shift amount 2672 which is usable to left-align close path preliminary result within left shift unit 2650.

In one embodiment, left shift unit 2650 is an n+1 bit shifter, where n is the width of close data path 2600 (32 bits in one embodiment). The shifter is configured to be n+1 bits in order to account for the one bit position prediction error which may occur using the T-G-Z methodology for leading 0/1 detection. All n+1 bits may thus be conveyed to final multiplexer unit 2500. If the most significant bit is set (indicating proper normalization), the most significant n bits of the n+1 bits conveyed to unit 250 are selected as the mantissa portion of result value 252. Conversely, if the most significant bit is not set, the least significant n bits of the n+1 bits conveyed to unit 2500 are selected as the mantissa portion of result value 252.

The exponent portion of close path result 242 is calculated by an exponent adjustment unit 2680 using either exponent large input 309 or the maximum exponent value for the given integer representation. For the 32-bit integer format described above, the maximum exponent value is 31 in one embodiment. This corresponds to the largest exponent possible for an integer value within the given format. The operation of adjustment unit 2680 is described below with reference to Fig. 35.

Turning now to Fig. 33, a block diagram of one embodiment of shift-swap unit 2610 is depicted. Shift-swap unit 2610 is similar to unit 710 described above with reference to Fig. 13. Unit 2610 is additionally configured, however, to select the proper operands for the i2f operation. As shown, unit 2610 is coupled to receive i2f signal 2602. In response to signal 2602 being asserted, input multiplexers 2702A is configured to output the zero operand (conveyed as the output of multiplexer 2601) as adder input 2612A, while input multiplexer 2702B is configured to output operand M_B. Operand M_B is then negated by inverter 2708 and conveyed as adder input 2612B.

Turning now to Fig. 34, a block diagram of one embodiment of multiplexer-inverter unit 2640 is depicted. Unit 2640 is similar in structure to unit 740 described above with reference to Fig. 17. Unit 2640 is additionally configured to provide proper selection for i2f conversions in addition to standard close path subtraction.

As shown, unit 2640 is coupled to receive adder outputs 2622A-B. For standard close path subtraction, close path select signal 2632 selects of one of the adder inputs to be conveyed as close path preliminary result 2642. Adder input 2622A may be inverted before selection by multiplexer 2800 for the case in which $E_A=E_B$ and the output of adder unit 2620 is negative.

The selection process for i2f conversion is similar. In one embodiment, selection unit 2630 generates close path select signal according to the sign of the integer input number is i2f signal 2602 is active. If the i2f input is a positive number, close path select signal 2632 is generated to be indicative of adder output 2622A (sum). Because a positive i2f input in close path 2600 produces a negative output from adder 2620 in one embodiment, proper recomplementation is provided by inverting adder output 2622A in XOR block 2810. This produces a result of the correct magnitude which may be conveyed as close preliminary result 2642. If, on the other hand, the i2f input is a negative number (expressed in two's complement form), selection of adder output 2622B by select signal 2632 produces a result of the correct magnitude. Sign bit logic (not shown) is also included in close data path 2600 to ensure that the target floating point number has the same sign as the input integer number.

Turning now to Fig. 35, a block diagram of one embodiment of exponent adjustment unit 2680 is depicted. As shown, unit 2680 includes an exponent multiplexer 2902, an inverter 2904, a shift count

adjustment multiplexer 29030, a half adder 2910, and a full adder 2920. Exponent adjustment unit 2680 is configured to subtract the predicted shift amount from an initial exponent in order to generate the exponent portion of close path result 242. In the case of standard close path subtraction (non-i2f operations), a correction factor is added back into the exponent to account for the difference in width between the integer and floating point formats. This function is described in greater detail below.

Consider an embodiment of close data path 2600 which is configured to handle a 32-bit integer format and a floating point format with a 24-bit mantissa portion. For standard close path subtraction, large exponent 309 is calculated within far data path 230 and conveyed to multiplexer 2902. Concurrently, predicted shift amount 2672 is calculated by shift prediction unit 2652 and conveyed to inverter 2904. The negated shift amount and large exponent 309 may then be added using half adder 2910 and full adder 2920. This adder configuration allows a correction constant conveyed from multiplexer 2930 to be added in as the second operand at bit 3 of full adder 2920. For standard close path operation, this constant is 1 (which is equivalent adding the value $2^3=8$ as a third operand to exponent adjustment calculation). The exponent adjustment calculation for standard close path subtraction becomes:

adjusted_exponent_value = expo_large - (shift_count - 8) (9);

adjusted_exponent_value = expo_large - shift_count + 8 (10).

This correction constant is used since standard close path subtractions are over-shifted by 8 bits by left shift unit 2650. Because shift prediction unit 2652 is configured to generate predicted shift amounts for both integer and floating point values within data path 2600, the shift amounts are based on left-aligning both sets of values with the larger format, which in this embodiment is the 32-bit integer format. Stated another way, normalizing the floating point values produced by close path subtraction only requires the MSB of the subtraction result to be left aligned with a 24-bit field. In order to accommodate 32-bit integers, however, all close path results are left-aligned with a 32-bit field. Accordingly, the predicted shift amount minus 8 is subtracted from large exponent 309 in order to produce the adjusted exponent. The carry in to bit 0 of full adder 2920 is set in order to compensate for the one's complement addition of shift amount 2672.

For i2f conversions, the exponent adjustment calculation is similar to that performed for standard close path subtraction. If i2f signal 2602 is active, however, the output of multiplexer 2902 is 31 and the correction constant conveyed from multiplexer 2930 is 0. Consider an i2f conversion in which the most significant bit of the adder output is located in bit 28 out of bits [31:0]. The floating point number resulting from this integer is $1.xxx \times 2^{28}$. The floating point exponent may thus be calculated by subtracting the shift amount (3) from the predetermined maximum integer exponent (31) without using a correction constant.

Although exponent adjustment unit 2680 is shown in Fig. 35 as being implemented with half adder 2910 and full adder 2920, various other adder configurations are also possible to produce the exponent portion of close path result 242.

As with the inclusion of floating point-to-integer conversion capability in far data path 2300, the expansion of close data path 2600 to handle integer-to-floating point conversion also provides extra versatility to add/subtract pipeline 220. The additional functionality is included within data path 2600 with a minimum

number of changes. Accordingly, i2f conversion capability is achieved with an efficient hardware implementation.

The embodiments shown above depict a single add/subtract pipeline 220 within each of execution units 136C and 136D. These embodiments allow concurrent execution of floating point add and subtract instructions, advantageously increasingly floating point performance. By configuring pipelines 220 to handle integer-to-float and float-to-integer conversions as described above, execution units 136C-D may concurrently perform these operations as well.

Performance may further be increased by configuring each of execution units 136C-D to include a plurality of add/subtract pipelines 220. As will be described below, this allows each of execution units 136C-D to perform vector operations (the ability the concurrently perform the same arithmetic/logical operations on more than one set of operands). This configuration also allows a number of other operations to be efficiently implemented by pipelines 220 at a small additional hardware cost. These instructions are particularly useful for the types of operations typically performed by units 136C-D.

Turning now to Fig. 36, a block diagram of one embodiment of execution unit 136C/D is depicted. As shown, execution unit 136C/D is coupled to receive operands 204A-D and an instruction indication 3002, and includes input unit 3010 and add/subtract pipelines 220A-B. Each of pipelines 220 includes a far and close data path which is configured to operate as described above. The outputs of each pipeline 220 is selected by one of result multiplexers 250. The outputs of multiplexers 250 are conveyed as result values 3008A-B for storage in output register 3006.

Instruction indication 3002 specifies which operation is performed concurrently in each pipeline 220. For example, if indication 3002 specifies an add operation, both pipelines 220 concurrently execute an add operation on operands 204. Pipeline 220A may add operands 204A and 204C, for instance, while pipeline 220B adds operands 204B and 204D. This operation is described in greater detail below. In one embodiment, indication 3002 may specify any of the instructions described below with reference to Figs. 37-49. Additional operand instruction information specifies the input values by referencing one or more storage locations (registers, memory, etc.).

As described above, add, subtract, float-to-integer, and integer-to-float conversion instruction may be performed in add/subtract pipeline 220 using far data path 230 and close data path 240. Vectored versions of these instructions for one embodiment of pipeline 220 are described below with reference to Figs. 37-42. The configuration of Fig. 36 with a plurality of pipelines 220 may additionally be expanded to handle a number of other vectored instructions such as reverse subtract, accumulate, compares, and extreme value instructions. Specific embodiments of such instructions are described with reference to Figs. 43-49. (Other embodiments of these instructions are also possible).

Turning now to Fig. 37A, the format of a vectored floating point add instruction ("PFADD") 3100 is shown according to one embodiment of microprocessor 100. As depicted, PFADD instruction 3100 includes an opcode value 3101 and two operand fields, first operand field 3102A and second operand field 3102B. The value specified by first operand field 3102A is shown as being "mmreg1", which, in one embodiment, maps to one of the registers on the stack of floating point execution unit 136E. In another embodiment, mmreg1 specifies a storage location within execution unit 136C or 136D or a location in main memory. The value

specified by second operand field 3102B is shown in one embodiment as either being another of the floating point stack registers or a memory location ("mmreg2/mem64"). Similarly, mmreg2 may also specify a register within execution unit 136C or 136D in another embodiment. As used in the embodiment shown in Fig. 36, operand fields 3102A-B each specify a pair of floating point values having a sign value, an exponent value, and a mantissa portion.

Turning now to Fig. 37B, pseudocode 3104 illustrating operation of PFADD instruction 3100 is given. As shown, upon execution of PFADD instruction 3100, a first vector portion (such as input value 204A in Fig. 36) of the value specified by first operand field 3102A is added to a first vector portion (e.g., 204C) of the input value specified by second operand field 3102B. As described above, this sum is computed within far path 230A of pipeline 220A. In the embodiment shown, this sum is then written back to the upper portion of operand 3102A (mmreg1[63:32]). In another embodiment of the instruction, a destination storage location may be specified which is different than either of the source operands.

PFADD instruction 3100 also specifies that a second vector portion of the input value specified by first operand field 3102A (e.g., 204B) is added to a second vector portion (e.g., 204D) of the input value specified by second operand field 3102B. This sum is computed in far data path 230B of add/subtract pipeline 220B. This sum is then written, in one embodiment, to the lower portion of the location specified by operand 3102A (mmreg1[31:0]), although an alternate destination location may be specified in another embodiment. In one embodiment, the two add operations specified by instruction 3100 are performed concurrently to improve performance.

Turning now to Fig. 38A, the format of a floating-point vectored subtract instruction ("PFSUB") 3110 is shown according to one embodiment of microprocessor 100. The format of PFSUB instruction 3110 is similar to that described above for PFADD instruction 3100. As depicted, PFSUB instruction 3110 includes an opcode value 3111 and two operands, first operand field 3112A and second operand field 3112B. The value specified by first operand field 3112A is shown as being "mmreg1", which, in one embodiment, maps to one of the registers on the stack of floating point execution unit 136E. In another embodiment, mmreg1 specifies a register or storage location within execution unit 136C/D. The value specified by second operand field 3112B is shown, in one embodiment, as either being another of the floating point stack registers or a memory location ("mmreg2/mem64"). Similarly, mmreg2 may also specify a register within execution unit 136C/D in another embodiment. As with PFADD instruction 3100, the values specified by operand fields 3112A-B for PFSUB instruction 3110 each specify a pair of floating point numbers each having a sign value, an exponent value, and a mantissa portion.

Turning now to Fig. 38B, pseudocode 3114 illustrating operation of PFSUB instruction 3110 is given. As shown, upon execution of PFSUB instruction 3110, a first vector portion (such as input value 204C shown in Fig. 36) of the input value specified by second operand field 3112B is subtracted from a first vector portion of the value (e.g., value 204A) specified by first operand field 3112A. As described above, this difference may be computed in either far path 230A or close path 240A of pipeline 220A depending on the exponent difference value between the operands. In the embodiment shown, this difference value is written back to the upper portion of the value specified by first operand field 3112A (mmreg1[63:32]), although an alternate destination may be specified in other embodiments.

PFSUB instruction 3110 also specifies that a second vector portion (such as value 204D) of the value specified by second operand field 3112B be subtracted from a second vector portion (e.g., 204B) of the input value specified by first operand field 3112A. This difference is written to the lower portion of operand 3112B (mmreg1[31:0]) in one embodiment, but may be written to another location in other embodiments. In a configuration such as that shown in Fig. 36, both difference calculations are performed concurrently in respective add/subtract pipelines 220 to improve performance.

Turning now to Fig. 39A, the format of a vectored floating point-to-integer conversion instruction ("PF2ID") 3120 is shown according to one embodiment of microprocessor 100. The format of PF2ID instruction 3120 is similar to those described above. As depicted, PF2ID instruction 3120 includes an opcode value 3121 and two operand fields, first operand field 3122A and second operand field 3122B. The value specified by first operand field 3122A is shown as being "mmreg1", which, in one embodiment, maps to one of the registers on the stack of floating point execution unit 136E. In another embodiment, mmreg1 specifies a register or storage location within one of execution units 136C-D. As will be described below, mmreg1 specifies a destination location for the result of instruction 3120. The value specified by second operand field 3122B is shown as either being another of the floating point stack registers or a memory location ("mmreg2/mem64"). (Operand field 3122B may also specify a register or storage location within one of execution units 136C-D). Operand field 3122B specifies a pair of floating point numbers having a sign value, an exponent value, and a mantissa portion. It is noted that instruction 3120 produces a pair of 32-bit signed integer values in the embodiment shown. A floating point-to-integer instruction which produces a pair of 16-bit signed integers is described below with reference to Figs. 40A-C.

Turning now to Fig. 39B, pseudocode 3124 for PF2ID instruction 3120 is given. In the embodiment described by pseudocode 3124, PF2ID instruction 3120 operates separately on the first and second floating point numbers specified by second operand field 3122B. If the first floating point number specified by operand 3122B is outside the allowable conversion range, the corresponding output value is clamped at either the maximum or minimum value. If the first floating point input value is within the allowable input range, a float-to-integer conversion is performed in far data path 220A as described above. In one embodiment, the resulting integer is written to the upper portion of the storage location specified by operand field 3122A. This storage location may map to a floating point register within execution unit 136E, or may alternately be located within execution unit 136C/D or in main memory.

Pseudocode 3124 also specifies a similar conversion process for the second floating point input value specified by operand field 3122B. This floating point number is converted to a signed 32-bit integer and written to the upper half of the storage location specified by operand field 3122A in one embodiment. If microprocessor 100 is configured to include a plurality of add/subtract pipelines 220, the second f2i conversion may be performed in add/subtract pipeline 220B concurrently with the first conversion to improve performance.

Turning now to Fig. 39C, a table 3128 is given illustrating the integer output values resulting from various floating point input values. It is noted that the f2i conversion process truncates floating point numbers, such that the source operand is rounded toward zero in this embodiment.

Turning now to Figs. 40A-C, the format and operation of another floating point-to-integer ("PF2IW") instruction 3130 is shown. PF2IW instruction 3130 includes an opcode 3131 and a pair of operands fields

3132A-B. Fig. 40-B gives pseudocode 3134 which describes the operation of PF2IW instruction 3130. Instruction 3130 operates in a similar fashion to instruction 3120 except that the target integers are signed 16-bit integers rather than signed 32-bit integers. The maximum and minimum values for instruction 3130 reflect this change. The f2i conversions are performed in far data paths 230A-B in the configuration of execution unit 136C/D shown in Fig. 36. Table 3138 shown in Fig. 40C illustrates the output values of instruction 3130 for various ranges of input values.

Turning now to Fig. 41A, the format of an integer-to-floating point ("PI2FD") instruction 3140 is given. Instruction 3140 includes an opcode value 3141 and a pair of operand fields 3142A-B. In the embodiment shown, instruction 3140 is usable to convert a pair of signed 32-bit integers (specified by operand field 3142B) to a pair of corresponding floating point numbers (specified by operand field 3142A). In other embodiments, instruction 3140 may be used to convert floating point numbers of other sizes.

Turning now to Fig. 41B, pseudocode 3144 illustrating operation of instruction 3140 is given. As shown, instruction 3140 performs integer-to-float conversions on each of the values specified by operand field 3142B. Using the execution unit 136C/D shown in Fig. 36, each of the conversions may be performed concurrently within close data paths 240A-B of add/subtract pipelines 220A-B.

Turning now to Figs. 42A-B, the format and operation of another integer-to-floating point ("PI2FW") instruction 3150 is shown. As depicted, instruction 3150 includes an opcode value 3151, and a pair of operand fields 3152A-B. In the embodiment shown, the source values are a pair of floating point numbers specified by operand field 3152B. Pseudocode 3154 given in Fig. 42B illustrates the operation of instruction 3150. Instruction 3150 operates similarly to PI2FD instruction 3140 described above with reference to Figs. 41A-B, but instruction 3150 converts a pair of 16-bit signed integers to corresponding floating point values. In one embodiment, these floating point output values are written to respective portions of the storage location specified by operand field 3152A.

Execution unit 136C/D shown in Fig. 36 is configured to handle vectored add, subtract, f2i, and i2f instructions as described above. As will be shown below, pipelines 220A-B may be enhanced to handle additional vectored instructions as well. These instructions include, but are not limited to, additional arithmetic instructions, comparison instructions, and extreme value (min/max) instructions. These instructions may be realized within pipelines 220 with relatively little additional hardware, yielding an efficient implementation. Specific embodiments of such instructions are described below with reference to Figs. 43-49, although other instruction formats are possible in other embodiments.

Turning now to Fig. 43A, the format of a floating point accumulate instruction ("PFACC") 3160 is shown according to one embodiment of the invention. As depicted, PFACC instruction 3160 includes an opcode value 3161 and two operand fields, first operand field 3162A and first operand field 3162B. First operand field 3162A ("mmreg1") specifies a first pair of floating point input values in one embodiment. Operand field 3162A may specify a location which maps to one of the registers on the stack of floating point execution unit 136E. In another embodiment, operand field 3162A specifies a register or storage location within execution unit 136C/D. Second operand field 3162B ("mmreg2") specifies a second pair of floating point input values. These input values may be located on the floating point stack of unit 136E or within a storage location in execution unit 136C/D.

Turning now to Fig. 43B, pseudocode 3164 illustrating operation of instruction 3160 is shown. Accumulate instruction 3160 is slightly different than other floating point vector operations described above (such as PFADD instruction 3100 and PFSUB instruction 3110). In the embodiments described above, instructions 3100 and 3110 operate on corresponding parts of two different register values to produce an output value. For example, PFADD instruction 3100 forms a first portion of a vector output value by adding a first vector portion of a first input register to a first vector portion of a second input register. In contrast, PFACC instruction 3160 adds the component values of each floating point input register separately. As shown in Fig. 43B, the first portion of the vector output value produced by instruction 3160 is equal to the sum of the pair of floating point input values within the storage location specified by first operand field 3162A. This addition operation is performed within far data path 230A of add/subtract pipeline 220A. The second portion of the vector output value for instruction 3160 is produced similarly within far data path 230B of add/subtract pipeline 220B.

Because PFACC instruction 3160 operates on vectored components of a single input storage location, this instruction is particularly advantageous in matrix multiply operations. Matrix multiply operations may be effectuated by performing vector multiply operations, then summing the resulting values to obtain a sum of products. It is noted that PFACC instruction 3160 provides an advantageous means for summing the result of these vector multiply operations, particularly if these results reside in a single vector register. Because matrix multiply operations are quite prevalent in 3-D graphics operations, the use of instruction 3160 may significantly increase the graphics processing capabilities (particularly with regard to front-end geometry processing) of a system which includes microprocessor 100.

Turning now to Fig. 44A, the format of a floating-point vectored reverse subtract instruction ("PFSUBR") 3170 is shown according to one embodiment of microprocessor 100. The format of PFSUBR instruction 3170 is similar to that described above for PFSUB instruction 3110. As depicted, PFSUBR instruction 3110 includes an opcode value 3171 and two operands, first operand field 3172A and second operand field 3172B. In a similar fashion to operands for instructions described above, the floating point input values specified by operand fields 3172A-B may map to the stack of floating point unit 136E in one embodiment. These values may additionally be located within a register or storage location within execution unit 136C/D.

It is noted that in the embodiment shown, the only difference between PFSUBR instruction 3170 and PFSUB instruction 3110 is the "direction" of the subtraction. In PFSUB instruction 3110, portions of the values specified by operand field 3112B are subtracted from corresponding portions of the values specified by operand field 3112A. Conversely, in PFSUBR instruction 3170, portions of the values specified by operand field 3172A are subtracted from the corresponding portions of the values specified by operand field 3172B.

Turning now to Fig. 44B, pseudocode 3174 illustrating operation of PFSUBR instruction 3170 is given. As shown, upon execution of PFSUBR instruction 3170, a first vector portion (such as input value 204A) of the value specified by first operand field 3172A is subtracted from a first vector portion (e.g., 204C) of the value specified by second operand field 3172B. This subtraction operation may either be performed within far data path 230A or close data path 240A depending upon the exponent difference value of the operands. In the embodiment shown, this difference value is written back to the upper portion of operand

3172A (mmreg1[63:32]). In other embodiments, the difference value may be written back to a different destination storage location. Concurrently, a second vector portion of the value specified by first operand field 302A is subtracted from a second vector portion of the value specified by second operand field 302B. This difference is written, in one embodiment, to the lower portion of the location specified by operand 302A (mmreg1[31:0]). In the configuration of execution unit 136C/D shown in Fig. 36, this second reverse subtract operation is performed either in far data path 230B or close data path 230B of add/subtract pipeline 220B.

The vectored floating point instructions described above are particularly useful in the geometry processing stages of a 3-D graphics pipeline. Another class of functions commonly utilized in graphics processing are extreme value functions. As used herein, "extreme value functions" are those functions which return as a result either a maximum or minimum value selected among a plurality of values. In typical multimedia systems, a minimum value or a maximum value is obtained through the execution of several sequentially executed instructions. For example, a compare instruction may first be executed to determine the relative magnitudes of a pair of operand values, and subsequently a conditional branch instruction may be executed to determine whether a move operation must be performed to move the extreme value to a destination register or other storage location. These sequences of commands commonly occur in multimedia applications, such as in clipping algorithms for graphics rendering systems. Since extreme value functions are implemented through the execution of multiple instructions, however, a relatively large amount of processing time may be consumed by such operations. Graphics processing efficiency may be advantageously increased by dedicated extreme value instructions as described below with reference to Figs. 45-46.

Turning now to Fig. 45A, the format of a floating point maximum value instruction ("PFMAX") 3180 is shown according to one embodiment of the invention. As depicted, PFMAX instruction 3180 includes an opcode value 3181 and two operands, first operand field 3182A and first operand field 3182B. The value specified by first operand field 3182A is shown as being "mmreg1", which, in one embodiment, is one of the registers on the stack of floating point execution unit 136E. As with operands described above for other instructions, the storage locations specified by operand field 3182A may be located in alternate locations such as execution unit 136C/D. Similarly, the values specified by second operand field 3182B, mmreg2, may also specify the floating point stack registers, a memory location, or a register within unit 136C/D. In another embodiment, second operand field 3182B specifies an immediate value.

Turning now to Fig. 45B, pseudocode illustrating operation of PFMAX instruction 3180 is given. As shown, upon execution of PFMAX instruction 3180, a comparison of a first vector portion (such as value 204A) of the value specified by first operand field 3182A and a first vector portion of the value specified by second operand 3182B (e.g., 204C) is performed. Concurrently, a comparison of a second vector portion (such as value 204B) of the value specified by first operand field 3182A and a second vector portion of the value specified by second operand field 3182B (e.g., 204D) is also performed.

If the first vector portion of the value specified by first operand field 3182A is found to be greater than the first vector portion of the value specified by second operand field 3182B, the value of the first vector portion of the value specified by first operand field 3182A is conveyed as a first portion of a result of instruction 3180. Otherwise, the value of the first vector portion of value specified by second operand field 3182B is conveyed as the first vector portion of the result of instruction 3180. The second vector portion of the result of

3182A 3182B
1st 204A ⇔ 204C
204B ⇔ 204D

the PFMAX instruction is calculated in a similar fashion using the second vector portions of the values specified by operands fields 3182A-B.

Turning now to Fig. 45C, a table 3188 is shown which depicts the output of instruction 3180 for various inputs. Table 3188 includes cases in which operands 3182 are set to zero or in unsupported formats.

Turning now to Figs. 46A-C, the format and operation of a vectored floating point ("PFMIN") instruction 3190 is shown. As depicted, instruction 3190 includes an opcode value 3191, and a pair of operands fields 3192A-B. Operation of PFMIN instruction 3190 is similar to that of PFMAX instruction 3180, although instruction 3190 performs a minimum value function instead of a maximum value function. The operation of instruction 3190 is given by pseudocode 3194 in Fig. 45B. Fig. 45C includes a table 3198 which illustrates outputs of PFMIN instruction 3190 for various input values, including zero values and unsupported formats.

As described above, vectored extreme value functions such as PFMAX instruction 3180 and PFMIN instruction 3190 are particularly useful for performing certain graphics processing functions such as clipping. Because the operands in extreme value functions are compared in order to produce a result value, vectored comparison instructions may also be realized within an execution unit 136C/D which is configured to perform extreme value instructions 3180 and 3190. Three such comparison instructions are described below with reference to Figs. 47-49.

Turning now to Fig. 47A, the format of a floating point equality compare instruction ("PFCMPEQ") 3200 is shown according to one embodiment of microprocessor 100. As depicted, PFCMPEQ instruction 3200 includes an opcode value 3201 and two operands, first operand field 3202A and first operand field 3202B. The value specified by first operand field 3202A is shown as being "mmreg1", which, in one embodiment, is one of the registers on the stack of floating point execution unit 136E. First operand field 3202A may also specify a register or storage location within execution unit 136C/D. The value specified by second operand field 3202B, "mmreg2", is shown as either being another of the floating point stack registers or a memory location. In another embodiment, second operand field 3202B specifies an immediate value or a register/storage location within unit 136C/D.

Turning now to Fig. 47B, pseudocode 3204 illustrating operation of PFCMPEQ instruction 3200 is given. As shown, upon execution of PFCMPEQ instruction 3200, a comparison of a first vector portion (such as value 204A) of the value specified by first operand field 3202A and a first vector portion of the value second operand 3202B (e.g., 204C) is performed. Concurrently, a comparison of a second vector portion (e.g., 204B) of the value specified by first operand field 3202A and a second vector portion of the value specified by second operand field 3202B (204D) is also performed.

If the first vector portion of the value specified by first operand field 3202A is found to be equal to the first vector portion of the value specified by second operand field 3202B, a first mask constant is conveyed as a first portion of a result of instruction 3200. In the embodiment shown, this first mask constant is all 1's (FFFF_FFFFh), but may be different in other embodiments. Otherwise, a second mask constant (0000_0000h in one embodiment) is conveyed as the first vector portion of the result of instruction 3200. Similarly, if the second vector portion of the value specified by first operand field 3202A is found to be equal to the second vector portion of the value specified by second operand field 302B, the first mask constant is conveyed as a second portion of a result of instruction 3200. Otherwise, the second vector portion of the result of instruction

3200 is conveyed as the second mask constant. Fig. 47C is a table which shows the output of instruction 3200 given various inputs, including cases in which operands 3202 are zero or in unsupported formats.

The result (both the first and second vector portions) of instruction 3200 is subsequently written to the storage location specified by operand field 3202A. In another embodiment of instruction 3200, the result value may be stored to mmreg2, a memory location, or a third register specified by an additional operand. It is noted that in other embodiments of operands 3202, these values may include additional vector values beyond the two vector values shown in Fig. 47A.

Turning now to Figs. 48A-C, the format and operation of a vectored floating point greater than compare operation ("PFCMPGT") instruction 3210 is shown. As depicted, instruction 3210 includes an opcode value 3211, and a pair of operand fields 3212A-B. Instruction 3210 is performed in a similar fashion to instruction 3200, although a greater than comparison test is performed instead of an equality test. The operation of PFCMPGT instruction 3210 is given by pseudocode listing 3214 in Fig. 48B. Fig. 48C includes a table 3218 which gives outputs for various input values of instruction 3210.

Processor Report

Turning now to Figs. 49A-C, the format and operation of a vectored floating point greater than or equal compare operation ("PFCMPGE") instruction 3220 is shown. As depicted, instruction 3220 includes an opcode value 3221, and a pair of operand fields 3222A-B. Instruction 3220 is performed in a similar fashion to instructions 3200 and 3210, although instruction 3220 effectuates a greater than or equal to comparison test. The operation of PFCMPGE instruction 3220 is given by pseudocode listing 3224 in Fig. 49B. Fig. 49C includes a table 3228 which gives outputs for various input values of instruction 3220.

Turning now to Fig. 50, a block diagram of another embodiment of execution unit 136C/D is shown. Like the embodiment shown in Fig. 36, execution unit 136C/D includes a pair of add/subtract pipelines 220A-B with respective far and close data paths for performing add, subtract, f2i, and i2f instructions as described above. The embodiment of execution unit 136C/D shown in Fig. 50, however, additionally includes an input unit 3310 and an output unit 3320 which allow implementation of a number of other instructions, particularly those described above with reference to Figs. 37-49.

As depicted, execution unit 136C/D is coupled to receive inputs into a pair of input registers 3304A-B. In one embodiment, each register 3304 is configured to store a first vector value and a second vector value. For example, input register 3304A is configured to store first vector portion 204A and second vector portion 204B. Similarly, input register 3304B is configured to store first vector portion 204C and second vector portion 204D. As described above, these registers may include either integer or floating point values depending upon the type of operation being performed.

The type of operation to be performed by execution unit 136C/D is conveyed by instruction indication 3302. Instruction indication 3302 may specify any number of operations, including those described above (add/subtract, accumulate, f2i, i2f, extreme value, compare). For the embodiment of execution unit 136C/D shown in Fig. 50, all of the instructions described above are performed. In alternate embodiments, a unit 136C/D may only execute a subset of these instructions. In still other embodiments, execution unit 136C/D may also execute additional instructions to those described above (a vectored floating point instruction which performs a less than comparison test, for example).

In response to receiving instruction indication 3302, input unit 3310 is configured to route the appropriate combination of operand values 204 to add/subtract pipelines 220A-B via operand buses 3012A-D. Each data path within each of pipelines 220A-B receives an "A" operand value and a "B" operand value, even if one or more of these values is not utilized within a particular data path. For example, an f2i instruction is performed in the far data path 230A of pipeline 220A in one embodiment. Accordingly, the values conveyed to close data path 230B in pipeline 220A are not utilized for that particular instruction. Furthermore, different portions of the A and B operands may be conveyed to data paths 230 and 240. As described above , in one embodiment, far data paths 230A-B receive full exponent values, while close data paths 240A-B receive only the two least significant bits of each exponent for performing leading 0/1 prediction.

With appropriate routing by input unit 3310, a number of similar arithmetic instructions may be performed within execution unit 136C/D with minimal additional overhead. Table 2 given below shows the routing of operands for various values of instruction indication 3302. It is noted that instruction indication 3302 may indicate an effective operation (e.g., effective addition or subtraction) rather than an explicit operation denoted by an opcode.

	Add/Subtract Pipeline 220A		Add/Subtract Pipeline 220B	
	Op A	Op B	Op A	Op B
PFADD	A_1	B_1	A_0	B_0
PFSUB	A_1	B_1	A_0	B_0
PFSUBR	A_0	B_0	A_1	B_1
PFACC	A_1	A_0	B_1	B_0
PF2ID, PF2IW	-	B_1	-	B_0
PI2FD, PI2FW	-	B_1	-	B_0

Table 2

With operands 204 appropriately routed to pipelines 220, far data paths 230A-B and close data paths 240A-B operate substantially as described above. Far data paths 230A-B perform effective addition, as well as effective subtraction for operands with $E_{diff}>1$. Conversely, close data paths 240A-B perform effective subtraction on operands with $E_{diff}\leq 1$. Each pipeline 220 selects its corresponding far path result 232 or close path result 242 to be conveyed as result value 252. Pipeline 220A generates result value 252A, while pipeline 220B generates result value 252B. Result values 252A-B are conveyed to output unit 3320 and utilized as described below to generate output values 3008A-B.

In addition to receiving result values 252A-B, output unit 3320 is coupled to receive a maximum integer value 3321, a minimum integer value 3322, first and second mask constants 3324A-B, and operands 204A-D (A_1, A_0, B_1, and B_0). Output unit 3320 includes clamping comparators 3030A-D, extreme value comparator 3340, output selection logic 3350, and output multiplexer 3360. Output multiplexer 3360 is configured to convey output values 3008A-B to output register 3006.

The values conveyed to the input of output multiplexer 3360 represent the possible outputs for all of the instructions described above with reference to Figs. 37-49. Result values 252A-B convey output values for add, subtract, f2i, i2f, and accumulate instructions. Maximum integer value 3321 and minimum integer value 3322 are used for clamping f2i instruction results if needed. Operand values 204A-D are used to generate the output of the extreme value (min/max) instructions. First and second mask constants 3324A-B are used as outputs of the comparison instructions such as the equality compare, greater than compare, and greater than or equal to compare instructions described above.

With the outputs for each of the instructions described above conveyed to output multiplexer 3360, output selection logic 3350 may be used to select the appropriate multiplexer 3360 inputs to be conveyed as output values 3308A-B. It is noted that because of the vector nature of the input and output registers of execution unit 136C/D, output multiplexer 3360 accordingly selects a pair of output values. Accordingly, multiplexer 3360 is shown in Fig. 50 as having sub-portion 3360A (configured to convey output 3308A) and sub-portion 3360B (configured to convey output 3308B). Output selection logic 3350 generates a pair of corresponding select signals, 3352A-B, to control each of these multiplexer sub-portions.

Output selection logic receives instruction indication 3302, the outputs of clamping comparators 3030A-D, and the output of extreme value comparator 3340. If instruction indication 3302 specifies that an arithmetic instruction is being performed, result values 252A-B are conveyed as output values 3008A-B to output register 3006.

If a floating point-to-integer instruction is specified by indication 3302, result values 252A and 252B (calculated in far data paths 230A-B, respectively) are conveyed as output values 3008A-B unless one or both values exceed maximum integer value 3321 or minimum integer value 3322. Overflow and underflow conditions are detected by clamping comparators 3330A-D and conveyed to output selection logic 3350. In one embodiment, the maximum and minimum integer values are conveyed as output values 3008 in place of the values which caused the overflow/underflow condition. The f2i instruction specified by indication 3302 may generate integers of a variety of sizes as described above.

If an integer-to-floating point instruction is specified by instruction indication 3302, result values 252A and 252B (calculated in close data paths 240A-B, respectively) are conveyed as output values 3008A-B. It is noted that in the embodiment shown, the dynamic range of the floating point format exceeds the maximum and minimum integer values, so overflow/underflow detection logic is not used for the i2f instruction. The i2f instruction may specify conversion of integers of a variety of sizes as described above.

Extreme value

If an extreme value instruction is indicated by instruction indication 3302, extreme value comparator 3350 generates a plurality of outputs usable to determine the maximum and minimum values from each input pair. For example, if instruction indication 3302 specifies a maximum value instruction, comparator 3350 tests whether operand 204A is greater than operand 204C. If operand 204A is greater, it is conveyed as output value 3008A. Otherwise, operand 204C is conveyed.

The outputs generated by comparator 3350 are also usable to implement the comparison instructions described above. If a comparison instruction is specified by indication 3302, comparator outputs 3350 determine whether first or second mask constant 3324 is conveyed for each output value 3008. It is noted that

different mask constants may be generated for each portion of output register 3006 depending upon the particular input values in question.

The embodiments of execution units 136C/D shown above provide an efficient means for performing floating point arithmetic operations such as add and subtract. The improved selection logic implemented in one embodiment of close path 240 results in an add/subtract pipeline 220 with only one full add and one full shift in each of data paths 230 and 240. Still further, data paths 230 and 240 may additionally be configured to perform floating point-to-integer and integer-to-floating point conversions with little additional hardware. Such a capability is particularly important for an embodiment of execution unit 136C/D which handles both integer and floating point data (which may or may not be vectored).

By including a plurality of add/subtract pipelines in execution units 136C and D, vectored floating point instructions may be performed. This capability is advantageous in applications such as geometry processing for graphics primitives, in which identical operations are performed repetitively on large sets of data. By configuring each of units 136C-D with a pair of add/subtract pipelines 220, up to four vectored floating point operations may be performed concurrently in microprocessor 100. By proper input multiplexing of input operands, execution unit 136C/D may be expanded to handle additional arithmetic operations such as reverse subtract and accumulate functions. Finally, proper output multiplexing allows execution unit 136C/D to accommodate additional instruction such as extreme value and comparison instructions.

Turning now to Fig. 51, a block diagram of one embodiment of a computer system 3400 including microprocessor 100 coupled to a variety of system components through a bus bridge 3402 is shown. Other embodiments are possible and contemplated. In the depicted system, a main memory 3404 is coupled to bus bridge 3402 through a memory bus 3406, and a graphics controller 3408 is coupled to bus bridge 3402 through an AGP bus 3410. Finally, a plurality of PCI devices 3412A-3412B are coupled to bus bridge 3402 through a PCI bus 3414. A secondary bus bridge 3416 may further be provided to accommodate an electrical interface to one or more EISA or ISA devices 3418 through an EISA/ISA bus 3420. Microprocessor 100 is coupled to bus bridge 3402 through a CPU bus 3424.

Bus bridge 3402 provides an interface between microprocessor 100, main memory 3404, graphics controller 3408, and devices attached to PCI bus 3414. When an operation is received from one of the devices connected to bus bridge 3402, bus bridge 3402 identifies the target of the operation (e.g. a particular device or, in the case of PCI bus 3414, that the target is on PCI bus 3414). Bus bridge 3402 routes the operation to the targeted device. Bus bridge 3402 generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus 3414, secondary bus bridge 3416 may further incorporate additional functionality, as desired. For example, in one embodiment, secondary bus bridge 3416 includes a master PCI arbiter (not shown) for arbitrating ownership of PCI bus 3414. An input/output controller (not shown), either external from or integrated with secondary bus bridge 3416, may also be included within computer system 3400 to provide operational support for a keyboard and mouse 3422 and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus 3424 between microprocessor 100 and bus bridge 3402 in other embodiments. Alternatively, the

external cache may be coupled to bus bridge 3402 and cache control logic for the external cache may be integrated into bus bridge 3402.

Main memory 3404 is a memory in which application programs are stored and from which microprocessor 100 primarily executes. A suitable main memory 3404 comprises DRAM (Dynamic Random Access Memory), and preferably a plurality of banks of SDRAM (Synchronous DRAM).

PCI devices 3412A-3412B are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device 3418 is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller 3408 is provided to control the rendering of text and images on a display 3426. Graphics controller 3408 may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory 3404. Graphics controller 3408 may therefore be a master of AGP bus 3410 in that it can request and receive access to a target interface within bus bridge 3402 to thereby obtain access to main memory 3404. A dedicated graphics bus accommodates rapid retrieval of data from main memory 3404. For certain operations, graphics controller 3408 may further be configured to generate PCI protocol transactions on AGP bus 3410. The AGP interface of bus bridge 3402 may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display 3426 is any electronic display upon which an image or text can be presented. A suitable display 3426 includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system 3400 may be a multiprocessing computer system including additional microprocessors (e.g. microprocessor 100a shown as an optional component of computer system 3400). Microprocessor 100a may be similar to microprocessor 100. More particularly, microprocessor 100a may be an identical copy of microprocessor 100. Microprocessor 100a may share CPU bus 3424 with microprocessor 100 (as shown in Fig. 51) or may be connected to bus bridge 3402 via an independent bus.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

Multi-function Bipartite Look-up Table

Turning now to Fig. 52, a block diagram of one embodiment of a microprocessor 10 is shown. As depicted, microprocessor 10 includes a predecode logic block 12 coupled to an instruction cache 14 and a predecode cache 15. Caches 14 and 15 also include an instruction TLB 16. A cache controller 18 is coupled to predecode block 12, instruction cache 14, and predecode cache 15. Controller 18 is additionally coupled to a bus interface unit 24, a level-one data cache 26 (which includes a data TLB 28), and an L2 cache 40. Microprocessor 10 further includes a decode unit 20, which receives instructions from instruction cache 14 and

predecode data from cache 15. This information is forwarded to execution engine 30 in accordance with input received from a branch logic unit 22.

Execution engine 30 includes a scheduler buffer 32 coupled to receive input from decode unit 20. Scheduler buffer 32 is coupled to convey decoded instructions to a plurality of execution units 36A-E in accordance with input received from an instruction control unit 34. Execution units 36A-E include a load unit 36A, a store unit 36B, an integer X unit 36C, an integer Y unit 36D, and a floating point unit 36E. Load unit 36A receives input from data cache 26, while store unit 36B interfaces with data cache 26 via a store queue 38. Blocks referred to herein with a reference number followed by a letter will be collectively referred to by the reference number alone. For example, execution units 36A-E will be collectively referred to as execution units 36.

Generally speaking, floating point unit 36E within microprocessor 10 includes one or more bipartite look-up tables usable to generate approximate output values of given mathematical functions. As will be described in greater detail below, these bipartite look-up tables are generated such that absolute error is minimized for table output values. In this manner, floating point unit 36E may achieve an efficient implementation of such operations as the reciprocal and reciprocal square root functions, thereby increasing the performance of applications such as three-dimensional graphics rendering.

In addition, floating point unit 36E within microprocessor 10 includes a multi-function look-up table usable to generate approximate output values of a plurality of given mathematical functions. As will be described in greater detail below, this multi-function look-up table is configured such that an efficient implementation of the look-up function is achieved for more than one mathematical function. In this manner, floating point unit 36E may increase the performance of such operations as the reciprocal and reciprocal square root functions, thereby enhancing three-dimensional graphics rendering capabilities of microprocessor 10.

In one embodiment, instruction cache 14 is organized as sectors, with each sector including two 32-byte cache lines. The two cache lines of a sector share a common tag but have separate state bits that track the status of the line. Accordingly, two forms of cache misses (and associated cache fills) may take place: sector replacement and cache line replacement. In the case of sector replacement, the miss is due to a tag mismatch in instruction cache 14, with the required cache line being supplied by external memory via bus interface unit 24. The cache line within the sector that is not needed is then marked invalid. In the case of a cache line replacement, the tag matches the requested address, but the line is marked as invalid. The required cache line is supplied by external memory, but, unlike the sector replacement case, the cache line within the sector that was not requested remains in the same state. In alternate embodiments, other organizations for instruction cache 14 may be utilized, as well as various replacement policies.

Microprocessor 10 performs prefetching only in the case of sector replacements in one embodiment. During sector replacement, the required cache line is filled. If this required cache line is in the first half of the sector, the other cache line in the sector is prefetched. If this required cache line is in the second half of the sector, no prefetching is performed. It is noted that other prefetching methodologies may be employed in different embodiments of microprocessor 10.

When cache lines of instruction data are retrieved from external memory by bus interface unit 24, this data is conveyed to predecode logic block 12. In one embodiment, the instructions processed by

microprocessor 10 and stored in cache 14 are variable-length (e.g., the x86 instruction set). Because decode of variable-length instructions is particularly complex, predecode logic 12 is configured to provide additional information to be stored in instruction cache 14 to aid during decode. In one embodiment, predecode logic 12 generates predecode bits for each byte in instruction cache 14 which indicate the number of bytes to the start of the next variable-length instruction. These predecode bits are stored in predecode cache 15 and are passed to decode unit 20 when instruction bytes are requested from cache 14.

Instruction cache 14 is implemented as a 32Kbyte, two-way set associative, writeback cache in one embodiment of microprocessor 10. The cache line size is 32 bytes in this embodiment. Cache 14 also includes a TLB 16, which includes 64 entries used to translate linear addresses to physical addresses. Many other variations of instruction cache 14 and TLB 16 are possible in other embodiments.

Instruction fetch addresses are supplied by cache controller 18 to instruction cache 14. In one embodiment, up to 16 bytes per clock cycle may be fetched from cache 14. The fetched information is placed into an instruction buffer that feeds into decode unit 20. In one embodiment of microprocessor 10, fetching may occur along a single execution stream with seven outstanding branches taken.

In one embodiment, the instruction fetch logic within cache controller 18 is capable of retrieving any 16 contiguous instruction bytes within a 32-byte boundary of cache 14. There is no additional penalty when the 16 bytes cross a cache line boundary. Instructions are loaded into the instruction buffer as the current instructions are consumed by decode unit 20. (Predecode data from cache 15 is also loaded into the instruction buffer as well). Other configurations of cache controller 18 are possible in other embodiments.

Decode logic 20 is configured to decode multiple instructions per processor clock cycle. In one embodiment, decode unit 20 accepts instruction and predecode bytes from the instruction buffer (in x86 format), locates actual instruction boundaries, and generates corresponding "RISC ops". RISC ops are fixed-format internal instructions, most of which are executable by microprocessor 10 in a single clock cycle. RISC ops are combined to form every function of the x86 instruction set in one embodiment of microprocessor 10.

Microprocessor 10 uses a combination of decoders to convert x86 instructions into RISC ops. The hardware includes three sets of decoders: two parallel short decoders, one long decoder, and one vectoring decoder. The parallel short decoders translate the most commonly-used x86 instructions (moves, shifts, branches, etc.) into zero, one, or two RISC ops each. The short decoders only operate on x86 instructions that are up to seven bytes long. In addition, they are configured to decode up to two x86 instructions per clock cycle. The commonly-used x86 instructions which are greater than seven bytes long, as well as those semi-commonly-used instructions are up to seven bytes long, are handled by the long decoder.

The long decoder in decode unit 20 only performs one decode per clock cycle, and generates up to four RISC ops. All other translations (complex instructions, interrupts, etc.) are handled by a combination of the vector decoder and RISC op sequences fetched from an on-chip ROM. For complex operations, the vector decoder logic provides the first set of RISC ops and an initial address to a sequence of further RISC ops. The RISC ops fetched from the on-chip ROM are of the same type that are generated by the hardware decoders.

In one embodiment, decode unit 20 generates a group of four RISC ops each clock cycle. For clock cycles in which four RISC ops cannot be generated, decode unit 20 places RISC NOP operations in the

remaining slots of the grouping. These groupings of RISC ops (and possible NOPs) are then conveyed to scheduler buffer 32.

It is noted that in another embodiment, an instruction format other than x86 may be stored in instruction cache 14 and subsequently decoded by decode unit 20.

Instruction control logic 34 contains the logic necessary to manage out-of-order execution of instructions stored in scheduler buffer 32. Instruction control logic 34 also manages data forwarding, register renaming, simultaneous issue and retirement of RISC ops, and speculative execution. In one embodiment, scheduler buffer 32 holds up to 24 RISC ops at one time, equating to a maximum of 12 x86 instructions. When possible, instruction control logic 34 may simultaneously issue (from buffer 32) a RISC op to any available one of execution units 36. In total, control logic 34 may issue up to six and retire up to four RISC ops per clock cycle in one embodiment.

In one embodiment, microprocessor 10 includes five execution units (36A-E). Store unit 36A and load unit 36B are two-staged pipelined designs. Store unit 36A performs data memory and register writes which are available for loading after one clock cycle. Load unit 36B performs memory reads. The data from these reads is available after two clock cycles. Load and store units are possible in other embodiments with varying latencies.

Execution unit 36C (Integer X unit) is a fixed point execution unit which is configured to operate on all ALU operations, as well as multiplies, divides (both signed and unsigned), shifts, and rotates. In contrast, execution unit 36D (Integer Y unit) is a fixed point execution unit which is configured to operate on the basic word and double word ALU operations (ADD, AND, CMP, etc.).

Execution units 36C and 36D are also configured to accelerate performance of software written using multimedia instructions. Applications that can take advantage of multimedia instructions include graphics, video and audio compression and decompression, speech recognition, and telephony. Units 36C-D are configured to execute multimedia instructions in a single clock cycle in one embodiment. Many of these instructions are designed to perform the same operation of multiple sets of data at once (vector processing). In one embodiment, unit 36C-D uses registers which are mapped on to the stack of floating point unit 36E.

Execution unit 36E contains an IEEE 754-compatible floating point unit designed to accelerate the performance of software which utilizes the x86 instruction set. Floating point software is typically written to manipulate numbers that are either very large or small, require a great deal of precision, or result from complex mathematical operations such as transcendentals. Floating point unit includes an adder unit, a multiplier unit, and a divide/square root unit. In one embodiment, these low-latency units are configured to execute floating point instructions in as few as two clock cycles.

Branch resolution unit 35 is separate from branch prediction logic 22 in that it resolves conditional branches such as JCC and LOOP after the branch condition has been evaluated. Branch resolution unit 35 allows efficient speculative execution, enabling microprocessor 10 to execute instructions beyond conditional branches before knowing whether the branch prediction was correct. As described above, microprocessor 10 is configured to handle up to seven outstanding branches in one embodiment.

Branch prediction logic 22, coupled to decode unit 20, is configured to increase the accuracy with which conditional branches are predicted in microprocessor 10. Ten to twenty percent of the instructions in

typical applications include conditional branches. Branch prediction logic 22 is configured to handle this type of program behavior and its negative effects on instruction execution, such as stalls due to delayed instruction fetching. In one embodiment, branch prediction logic 22 includes an 8192-entry branch history table, a 16-entry by 16 byte branch target cache, and a 16-entry return address stack.

Branch prediction logic 22 implements a two-level adaptive history algorithm using the branch history table. This table stores executed branch information, predicts individual branches, and predicts behavior of groups of branches. In one embodiment, the branch history table does not store predicted target addresses in order to save space. These addresses are instead calculated on-the-fly during the decode stage.

To avoid a clock cycle penalty for a cache fetch when a branch is predicted taken, a branch target cache within branch logic 22 supplies the first 16 bytes at that address directly to the instruction buffer (if a hit occurs in the branch target cache). In one embodiment, this branch prediction logic achieves branch prediction rates of over 95%.

Branch logic 22 also includes special circuitry designed to optimize the CALL and RET instructions. This circuitry allows the address of the next instruction following the CALL instruction in memory to be pushed onto a return address stack. When microprocessor 10 encounters a RET instruction, branch logic 22 pops this address from the return stack and begins fetching.

Like instruction cache 14, L1 data cache 26 is also organized as two-way set associative 32Kbyte storage. In one embodiment, data TLB 28 includes 128 entries used to translate linear to physical addresses. Like instruction cache 14, L1 data cache 26 is also sectored. Data cache 26 implements a MESI (modified-exclusive-shared-invalid) protocol to track cache line status, although other variations are also possible. In order to maximize cache hit rates, microprocessor 10 also includes on-chip L2 cache 40 within the memory sub-system.

Turning now to Fig. 53, a graph 50 of a function f(x) is depicted which corresponds to a prior art look-up table described below with reference to Fig. 54. Graph 50 includes a portion 80 of function f(x), with output values 82A-E plotted on a vertical axis 60 against corresponding input values on a horizontal axis 70.

As will be described below, a look-up table for function f(x) is designed by dividing a predetermined input range into one or more various sub-regions. A single value is generated for each of the one or more sub-regions, and then stored into the look-up table. When an input value is presented to the look-up table, an index is formed which corresponds to one of the sub-regions of the input range. This index is then usable to select one of the predetermined output values.

In Fig. 53, input range portion 64 corresponds to portion 80 of function f(x). As shown, input range 64 is divided into a plurality of intervals 72. Interval 72A, for example, corresponds to input values located between points 71A and 71B on the horizontal axis. Interval 72B corresponds to input values located between points 71B and 71C, etc. It is noted that while only four intervals are shown in graph 50, many intervals are typically computed for a given function. Only four are shown in Fig. 53 for simplicity.

As mentioned, each interval 72 has a corresponding range of output values. Interval 72A, for example, includes a range of output values spanning between points 82A and 82B. In order to construct a look-up table for function f(x), a single output value is selected for interval 72A which has a value between points 82A and 82B. The method of selecting this output value varies between look-up tables. The method used for selecting

output values for various input sub-regions in one embodiment of the present invention is described in detail below.

Turning now to Fig. 54, a block diagram of a prior art look-up table 100 is depicted. Look-up table 100 is configured to receive an input value 102 and generate an output value 112. Input value 102 is conveyed to an address control unit 104, which in turn generates an index 106 to a table portion 108. Table portion 108 includes a plurality of table entries 110. Index 106 selects one of table entries 110 to be conveyed as output value 112.

The implementation of look-up table 100 is advantageous for several reasons. First, index 106 is readily generated from input value 102. Typically, input value 102 is represented in binary format as a floating point number having a sign bit, a mantissa portion, and an exponent. Index 106, then, is formed by selecting a sufficient number of high-order mantissa bits to table portion 108, which usually includes a number of entries 2^m, where m is some integer value. For example, if table portion 108 includes 64 entries, six high-order bits from the mantissa portion of input value 102 are usable as index 106. Another advantage of look-up table 100 is that output value 112 is usable as a output value of function f(x) without the additional step of interpolation (which is used in other look-up tables described below).

No interpolation is needed because input range portion 24 (and any additional range of input values) is divided into intervals for which a single output value is assigned. Each table entry 110 corresponds to one of these intervals as shown in Fig. 54. For example, table entry 110A corresponds to interval 32A, table entry 110B corresponds to interval 32B, etc. With this configuration, in order to increase the accuracy of output value 112, the number of intervals 32 are increased. This decreases the range of input values in each interval, and hence, the maximum possible error. Since a table entry 110 is provided for each interval 32, an increase in the number of intervals leads to a corresponding increase in table size. (Table size is equal to $P*2^{index}$ bits, where P is the number of bits per table entry, and 2^{index} is the number of table entries.) For many functions, in order to achieve the desired degree of accuracy, the input range is divided into a large number of intervals. Since there is a one-to-one correspondence between the number of intervals 32 and the number of table entries 110, achieving the desired degree of accuracy for many functions may lead to a prohibitively large look-up table.

Turning now to Fig. 55, a graph 120 is depicted of a portion 150 of function f(x). The partitioning of function portion 150 corresponds to a prior art look-up table described below with reference to Fig. 56. Graph 120 includes a portion 150 of function f(x), with output values 152A-E plotted on a vertical axis 130 against corresponding input values on a horizontal axis 140.

Fig. 55 illustrates a different input range partitioning for function f(x) than is shown in Fig. 53. This partitioning allows an interpolation scheme to be implemented for the look-up table described below with reference to Fig. 56. The input range of function f(x) is, as above, divided into intervals. Intervals 142A and 142B are shown in Fig. 55, although a given function may have any number of intervals depending upon the particular embodiment. Each interval 142 is then divided into subintervals. Interval 142A, for example, is divided into subintervals 144A-D, while interval 142B is divided into subintervals 146A-D.

With the input range of function f(x) partitioned as shown, a bipartite table look-up may thus be constructed which includes separate base and difference portions. The base portion of the bipartite look-up

table includes an output value for each interval 142. The output value is located somewhere within the range of output values for the interval. For example, the output value selected for interval 142A is located between points 152A and 152E. Which subinterval 144 the base value for interval 142A is located in depends upon the particular embodiment.

The difference portion of the bipartite look-up table includes an output value difference for each subinterval. This output value difference may then be used (along with the base value for the interval) to compute an output of the bipartite look-up table. Typically, the output value difference is either added to the base value or subtracted from the base value in order to generate the final output.

For example, consider this method as applied to interval 142. First, an output value is chosen to represent each subinterval 144. Then, an output value is chosen for the entire interval 142A. In one embodiment, the chosen output value for interval 142A may be identical to one of the output values chosen to represent one of subintervals 144. The output value chosen to represent interval 142A is then used as the corresponding base portion value. The differences between this base portion value and the values chosen to represent each of subintervals 144 are used as the difference portion entries for interval 142A.

Turning now to Fig. 56, a block diagram of a prior art look-up table 200 is depicted. Look-up table 200 is configured to receive an input value 202 and generate an output value 232. Input value 202 is conveyed to an address control unit 210, which in turn generates a base table index 212 and a difference table index 214. Base table index 212 is conveyed to a base table 220, while difference table index 214 is conveyed to a difference table 224. Base table 220 includes a plurality of table entries 222. Base table index 212 selects one of entries 222 to be conveyed to an output unit 230 as a base table value 223. Similarly, difference table 224 includes a plurality of entries 226. Difference table index 214 selects one of entries 226 to be conveyed to output unit 230 as a difference table value 227. Output unit 230 then generates output value 232 in response to receiving base table value 223 and difference table value 227.

The indexing scheme of look-up table 200 is only slightly more complicated than that of look-up table 100. Similar to index 106, base table index 212 is formed by a number of high-order mantissa bits in the binary representation of input value 202. Like table portion 108, base table 220 includes an entry 222 for each interval 142 in the predetermined input range of function f(x). Typically there are 2^{index} entries, where index is the number of bits in base table index 212. The bits of index 212 plus an additional number of bits are used to form index 214. If the number of subintervals per interval, s, is a power of two, this number of additional bits is equal to $\log_2 s$. In general, the number of additional bits is sufficient to specify all subintervals per interval s.

This implementation may result in a savings of table storage for table 200 with respect to table 100. Consider intervals 32A-D of Fig. 53. In table 100, entries in table portion 108 each include P bits. Thus, the storage requirement for these four intervals is 4*P bits in a scheme in which no interpolation is utilized. With the intervals 32A-D partitioned as in Fig. 55, however, intervals 32A-D become a single interval having four subintervals. The storage requirements for this partitioning would be a single base table entry 222 of P bits (for the one interval) and four difference table entries 226 (one per subinterval) of Q bits each. For this example, then, the total storage requirement for this bipartite scheme is P + 4*Q bits, where Q is the number of bits in each difference entry. If Q is sufficiently smaller than P, the bipartite implementation of table 200 results in a reduced storage requirement vis-a-vis table 100. This condition is typically satisfied when function f(x)

changes slowly, such that few bits are required to represent the difference values of difference table 224. Note that the above example is only for a single interval of a given function. In typical embodiments of look-up tables, function input ranges are divided into a large number of input sub-regions, and table size savings is applicable over each of these sub-regions.

Turning now to Fig. 57, a graph 250 of a function f(x) is depicted which corresponds to a look-up table according to one embodiment of the present invention. This look-up table is described below with reference to Fig. 58. Graph 250 includes a portion 280 of function f(x), with output values 282A-Q plotted on a vertical axis 260 against corresponding input values x on a horizontal axis 270.

Fig. 57 depicts yet another partitioning of the range of inputs for function f(x). This partitioning allows an interpolation scheme to be implemented for the look-up table of Fig. 58 which allows further reduction in table storage from that offered by the configuration of table 200 in Fig. 56. The input range of function f(x) is, as above, divided into intervals. Only one interval, 272A, is shown in Fig. 57 for simplicity, although a given function may have any number of intervals, depending upon the embodiment. As shown, interval 272A is divided into a plurality of subintervals 274A-D. Additionally, each subinterval 274 is divided into a plurality of sub-subintervals. Subinterval 274A is divided into sub-subintervals 276A-D, subinterval 274B is divided into sub-subintervals 277A-D, etc.

With the partitioning shown in Fig. 57, a bipartite look-up table 300 may be constructed which is similar to table 200 shown in Fig. 56. Table 300 is described in detail below with reference to Fig. 58. Like table 200, table 300 includes a base table portion and a difference table portion. The entries of these tables, however, correspond to regions of the input range of function f(x) in a slightly different manner than the entries of table 200. The base table portion of table 300 includes an entry for each subinterval in the input range. Each base table entry includes a single output value to represent its corresponding subinterval. The base table entry for subinterval 274A, for example, is an output value between those represented by points 282A and 282E. Instead of including a separate difference table entry for each sub-subinterval in each subinterval, however, table 300 has a number of difference table entries for each interval equal to the number of sub-subintervals per subinterval. Each of these entries represents an averaging of difference values for a particular group of sub-subintervals within the interval.

Consider the partitioning shown in Fig. 57. An output value is determined for each subinterval 274, and each sub-subinterval 276-279. As will be described below, in one embodiment of the present invention, the output value for each subinterval and sub-subinterval is chosen such that maximum possible absolute error is minimized for each input region. The base table entries are computed by using the assigned output value for each of subintervals 274. A separate entry is entered for each of regions 274A-D. Then, difference values are computed for each sub-subinterval which are equal to the difference between the output value for the sub-subinterval and the output value assigned for the subinterval. Then, the difference values are averaged for sub-subintervals having common relative positions within the subintervals. These values are then used as the difference table entries.

For example, difference values are computed for each of sub-subintervals 276-279 and their respective subintervals. Then difference values for sub-subintervals 276A, 277A, 278A, and 279A are averaged to form the first difference entry for interval 272. Difference values for sub-subintervals 276B, 277B, 278B, and 279B

are averaged to form the second difference entry, etc. This results in a number of difference entries per interval equal to the number of sub-subintervals per interval.

Like table 200, the base and difference table values may be combined to form a final output value. While the configuration of table 300 may result in a reduced table size, a slight increase in the number of bits in each table may be needed in order to achieve the same result accuracy as table 200.

Turning now to Fig. 58, a block diagram of look-up table 300 is depicted according to one embodiment of the present invention. Look-up table 300 is configured to receive an input value 302 and generate an output value 332. Input value 302 is conveyed to an address control unit 310, which in turn generates a base table index 312 and a difference table index 314. Base table index 312 is conveyed to a base table 320, while difference table index 314 is conveyed to a difference table 324. Base table 320 includes a plurality of table entries 322. Base table index 312 selects one of entries 322 to be conveyed to an output unit 330 as a base table value 323. Similarly, difference table 324 includes a plurality of entries 326. Difference table index 314 selects one of entries 326 to be conveyed to output unit 230 as difference table value 327. Output unit 330 then generates output value 332 in response to receiving base table value 323 and difference table value 327.

The indexing scheme of look-up table 300 is slightly different than that used to address table 200. In one embodiment, three groups of bits from a binary representation of input value 302 are used to generate indices 312 and 314. The first group includes a number of high-order mantissa bits sufficient to uniquely specify each interval of the input range of function f(x). For example, the first group includes four bits if the input range of function f(x) is divided into 16 intervals. Similarly, the second bit group from the binary representation of input value 302 has a number of bits sufficient to uniquely specify each subinterval included within a given interval. For example, if each interval includes four subintervals (such as is shown in Fig. 57), the second bit group includes two bits. Finally, the third bit group includes a number of bits sufficient to uniquely identify each group of sub-subintervals within a given interval. In this context, a group of sub-subintervals includes one sub-subinterval/subinterval, with each sub-subinterval in the group having the same relative position within its respective subinterval. The third bit group thus includes a number sufficient to specify the number of sub-subintervals in each subinterval. For the partitioning shown in Fig. 57, two bits are needed in the third bit group in order to specify each group of sub-subintervals. This addressing scheme is described in greater detail below.

Because base table 320 includes an entry for each subinterval in the input range of function f(x), base table index 312 includes the first and second bit groups described above from the binary representation of input value 302. Base table index 312 is thus able to select one of entries 322, since the first bit group effectively selects an input interval, and the second bit group selects a subinterval within the chosen interval. As shown in Fig. 58, each of table entries 322A-D corresponds to a different subinterval 274 within interval 272A.

Difference table 324 includes a set of entries for each interval equal to the number of sub-subintervals per subinterval. As shown, difference table 324 includes four entries 326 for interval 272A. Entry 326A corresponds to sub-subintervals 276A, 277A, 278A, and 279A, and includes an average of the actual difference values of each of these sub-subintervals. Difference table index 314 thus includes the first and third bit groups described above from the binary representation of input value 302. The first bit group within index 314

effectively selects an interval within the input range of function f(x), while the third bit group selects a relative position of a sub-subinterval within its corresponding subinterval.

The configuration of table 300 may result in a savings in table storage size with respect to tables 100 and 200. Consider the partitioning of function portion 280 shown in graph 250. Function portion 280 is divided into 16 equal input regions (called "sub-subintervals" with reference to Fig. 58).

In the configuration of table 100, the 16 input regions of Fig. 57 correspond to intervals. Each of the 16 intervals has a corresponding entry of P bits in table portion 108. Thus, the partitioning of Fig. 57 results in a table size of 16*P bits for the configuration of table 100.

By contrast, in the configuration of table 200, the 16 input regions in Fig. 57 would represent intervals divided into subintervals. In one embodiment, the 16 input regions are divided into four intervals of four subintervals each. Each interval has a corresponding entry of P bits in base table 220, while each of the 16 subintervals has a difference entry of Q bits in difference table 224. For this partitioning, then, the table storage size of table 200 is 4*P + 16*Q bits. The configuration of table 200 thus represents a storage savings over table 100 if function f(x) changes slowly enough (Q is greater for functions with steeper slopes, since larger changes are to be represented).

The configuration of table 300 represents even greater potential storage savings with respect to tables 100 and 200. As shown in Fig. 58, function portion 280 includes an interval 272A divided into four subintervals 274. Each subinterval 274 is divided into sub-subintervals, for a total of 16 input regions. Each subinterval has a corresponding entry of P' bits in base table 320 (P' is potentially slightly larger than P in order to achieve the same degree of accuracy). For interval 272A, difference table 224 has four entries of Q' bits each (Q' is potentially slightly larger than Q since averaging is used to compute the difference values). The total table storage requirement for table 300 is thus 4*P' + 4*Q' bits. Depending on the slope of function f(x), this represents a potential savings over both tables 100 and 200. The configuration of table 300 is well-suited for large, high-precision tables.

Turning now to Fig. 59, a format 400 for input values used in one embodiment of the invention is illustrated. Generally speaking, look-up tables according to the present invention are compatible with any binary floating-point format. Format 400 (the IEEE single-precision floating-point format) is one such format, and is used below in order to illustrate various aspects of one embodiment of the invention.

Format 400 includes a sign bit 402, an 8-bit exponent portion 404, and a 23-bit mantissa portion 406. The value of sign bit 402 indicates whether the number is positive or negative, while the value of exponent portion 404 includes a value which is a function of the "true" exponent. (One common example is a bias value added to the true exponent such that all exponent 404 values are greater than or equal to zero). Mantissa portion 406 includes a 23-bit fractional quantity. If all table inputs are normalized, values represented in format 400 implicitly include a leading "1" bit. A value represented by format 400 may thus be expressed as

$$x = (-1)^s \cdot 2^{expo} \cdot mant \,, \qquad (1)$$

where s represents the value sign bit 402, *expo* represents the true exponent value of the floating point number (as opposed to the biased exponent value found in portion 404), and *mant* represents the value of mantissa portion 406 (including the leading one bit).

An important floating-point operation, particularly for 3-D graphics applications, is the reciprocal function (1/x), which is commonly used during the perspective division step of the graphics pipeline. The reciprocal function may be generally expressed as follows:

$$\frac{1}{x} = \frac{1}{(-1)^s \cdot 2^{expo} \cdot mant}, \qquad (2)$$

$$\text{or } \frac{1}{x} = \frac{1}{(-1)^s} \cdot \frac{1}{2^{expo}} \cdot \frac{1}{mant}, \qquad (3)$$

which simplifies to

$$\frac{1}{x} = (-1)^s \cdot 2^{-expo} \cdot \frac{1}{mant} \text{ or} \qquad (4a)$$

$$\frac{1}{x} = (-1)^s \cdot 2^{-1-expo} \cdot \frac{2}{mant}. \qquad (4b)$$

Since the reciprocal of *mant* is clearly the difficult part of the operation, it is advantageous to implement an approximation to this value using table look-up. Since table input values (e.g., input value 302) are normalized, *mant* is restricted to

$$2^N \leq mant < 2^{N+1}, \qquad (5)$$

for some fixed N. In order to compute the reciprocal of all floating-point numbers, then, it suffices to compute 1/*mant* over the primary range $[2^N, 2^{N+1})$, and map all other inputs to that range by appropriate exponent manipulation (which may be performed in parallel with the table look-up).

Another common graphics operation is the reciprocal square root operation ($x^{-1/2}$), used in distance and normalization calculations. Defining sqrt(-x) = -sqrt(x) in order to handle negative inputs, this function may be expressed as follows:

$$\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{(-1)^s \cdot 2^{expo} \cdot mant}}, \text{ or} \quad (6)$$

$$\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{(-1)^s}} \cdot \frac{1}{\sqrt{2^{expo}}} \cdot \frac{1}{\sqrt{mant}}, \quad (7)$$

which simplifies to

$$\frac{1}{\sqrt{x}} = (-1)^s \cdot 2^{-\left(\frac{expo}{2}\right)} \cdot \frac{1}{\sqrt{mant}}. \quad (8)$$

Because having the exponent of 2 be a whole number in equation (8) is desirable, the reciprocal square root function may be written as two separate equations, depending upon whether *expo* is odd or even. These equations are as follows:

$$\frac{1}{\sqrt{x}} = (-1)^s \cdot 2^{\left(-\frac{expo}{2}\right)} \cdot \frac{1}{\sqrt{mant}} \; (expo \text{ even}) \; (9), \text{ and}$$

$$\frac{1}{\sqrt{x}} = (-1)s \cdot 2^{\left(-\frac{expo-1}{2}\right)} \cdot \frac{1}{\sqrt{2 \cdot mant}} \; (expo \text{ odd}) \; (10).$$

As with the reciprocal function, the difficult part of the reciprocal square root function is the computation of 1/sqrt(*mant*) or 1/sqrt(2**mant*). Again, this is implemented as a table look-up function. From equations (9) and (10), it can be seen that in one embodiment of a look-up table for the reciprocal square root function, the look-up table inputs may span two consecutive binades in order to handle both odd and even exponents. For true exponent values that are even, then, the input range is $[2^N, 2^{N+1})$, with odd true exponent values occupying the next binade, $[2^{N+1}, 2^{N+2})$.

It is noted that the order of the binades may be reversed for a look-up table that receives biased exponent values with a format that has an odd bias value. Thus, the lower half of a look-up table for the reciprocal square root function may contain entries for the binade defined by [2,4), while the upper order addresses include entries for the binade [1,2). Alternatively, the least significant bit of the biased exponent value may be inverted so that binade [1,2) entries are in the lower half of the look-up table.

For any binary floating-point format (such as format 400), a table look-up mechanism may be constructed for the reciprocal and reciprocal square root functions by extracting some number IDX of high-order bits of mantissa portion 406 of the input value. The look-up table includes P bits for each entry, for a total size (in a naïve implementation) of $P*2^{IDX}$ bits. The computation of the output sign bit and the output exponent portion are typically computed separately from the table look-up operation and are appropriately combined with

the table output to generate the output value (be it a reciprocal or a reciprocal square root). Note that since the numeric value of each mantissa bit is fixed for a given binade, extracting high-order bits automatically ensures equidistant nodes over the binade, such that interpolation may be performed easily.

As described above, the table look-up mechanism for the reciprocal square root has input values ranging over two consecutive binades. If it is desired to have equidistant nodes across both binades, IDX high-order bits may extracted from mantissa value 406 for the lower binade, with IDX+1 bits extracted from value 406 for the upper binade (this is done since the numeric value of each fractional bit in the upper binade is twice that of the same bit in the lower binade). In this implementation, the reciprocal square root function has a storage size of $P*2^{IDX}+P*2^{IDX+1}=3*P*2^{IDX}$ bits. In one embodiment, the required table accuracy allows table size to be reduced to $2*P*2^{IDX}=P*2^{IDX+1}$ bits by always extracting IDX leading fractional mantissa bits for each binade. This results in reducing the distance between the nodes in the upper binade. For the reciprocal square root function (1/sqrt(x)), the slope decreases rapidly for increasing x, which offsets table quantization error in the upper binade. Thus, nodes in a given binade (either upper or lower) are equidistant, but the distance between nodes varies in adjacent binades by a factor of two.

In one embodiment, performing table look-up for the reciprocal square root function may be accomplished by making one table for each of the two binades and multiplexing their output based upon the least significant bit of the value of exponent portion 404. In another embodiment, a single table may be implemented. This single table is addressed such that the IDX leading fractional bits of mantissa value 406 constitute bits <(IDX-1):0> of the address, with the least significant bit of exponent value 404 bit <IDX> of the table address. Such a table is discussed in greater detail below.

Turning now to Fig. 60A, a look-up table input value 420 according to format 400 is depicted. Input value 420 includes a sign bit (IS) 422, an exponent value (IEXPO) 424, and a mantissa value (IMANT) 426. In the embodiment shown, input value 420 is normalized, and mantissa value 426 does not include the leading one bit. Accordingly mantissa value 426 is shown as having N-1 bits (mantissa value 426 would be shown as having N bits in an embodiment in which the leading one bit is stored explicitly). The most significant bit in mantissa value 426 is represented in Fig. 60A as IMANT<N-2>, while the least significant bit is shown as IMANT<0>.

Turning now to Fig. 60B, an exploded view of mantissa value 426 is shown according to one embodiment of the present invention. In one embodiment, the bits of mantissa value 426 may be grouped according to the scheme shown in Fig. 60B in order to index into base and difference table portions of a look-up table for the reciprocal function. Other bit grouping are possible in alternate embodiments of the present invention.

The first group of bits is XHR 430, which is HR consecutive bits from IMANT<N-2> to IMANT<N-1-HR>. Similarly, the second group of bits is XMR 432, which includes MR consecutive bits from position IMANT<N-2-HR> to IMANT<N-1-HR-MR>, while the third group of bits, XLR 434, includes LR consecutive bits from IMANT<N-2-HR-MR> to IMANT<N-1-HR-MR-LR>. As will be described below, XHR 430 is used to specify the interval in the input range which includes the input value. Likewise, XMR 432 identifies the subinterval, and XLR the sub-subinterval group.

In one embodiment, the input value range for the reciprocal function for which look-up values are computed is divided into a plurality of intervals, each having a plurality of subintervals that are each divided into a plurality of sub-subintervals. Accordingly, XHR 430, XMR 432, and XLR 434 may each be as short as one bit in length (although the representation in Fig. 60B shows that each bit group includes at least two bits). Because each of these quantities occupies at least one bit in mantissa value 426, none of bit groups 430, 432, and 434 may be more than N-3 bits in length.

Turning now to Fig. 60C, a reciprocal base table index 440 is shown. As depicted, index 440 is composed of bit group XHR 430 concatenated with bit group XMR 432. As will be described below, index 440 is usable to select a base entry in a bipartite look-up table according to one embodiment of the present invention. In one embodiment, XHR 430 includes sufficient bits to specify each interval in the input range, while XMR 432 includes sufficient bits to specify each subinterval within a given interval. Accordingly, index 440 is usable to address a base table portion which includes an entry for each subinterval of each interval.

Turning now to Fig. 60D, a reciprocal difference table index 450 is shown. As depicted, index 450 is composed of bit group XHR 430 concatenated with bit group XLR 434. As will be described below, index 450 is usable to select a difference entry in a bipartite look-up table according to one embodiment of the present invention. As described above, XHR 430 includes sufficient bits to specify each interval in the input range, while XLR 432 includes sufficient bits to specify a group of sub-subintervals within a given interval. (As stated above, each group of sub-subintervals includes one sub-subinterval per subinterval, each sub-subinterval having the same relative position within its respective subinterval). Accordingly, index 450 is usable to address a difference table portion which includes an entry for each sub-subinterval group of each interval.

Turning now to Fig. 61A, mantissa value 426 is shown with different groupings of bits. Mantissa value 426 is partitioned in this manner when input value 420 corresponds to a second function, the reciprocal square root. The base and difference indices generated from the bit groupings of Fig. 61A are usable to obtain base and difference values for the reciprocal square root function within a bipartite look-up table according to one embodiment of the present invention.

Like the groupings of Fig. 60B, mantissa value 426 includes a first bit group XHS 460 which includes HS bits. This first group is followed by a second bit group XMS 462, having MS bits, and a third bit group XLS 464, with LS bits. In one embodiment, groups 460, 462, and 464 have the same length restrictions as groups 430, 432, and 434.

Fig. 61A is illustrative of the fact that the indices for each function in a multi-function bipartite look-up table do not have to be identical. Instead, the indices may be adjusted according to how the individual input ranges for the different functions are partitioned. For example, in one embodiment, a bipartite look-up table may include base and difference values for a first and second function. If greater accuracy is required for the second function in comparison to the first function, the input range of the second function may be partitioned differently than that of the first (the second function input range may be divided into more intervals, subintervals, etc.). Accordingly, this leads to more bits in the base and difference table indices for the second function. As will be shown below, however, it is often advantageous for the base and difference table indices to be identical in length (HR=HS, MR=MS, and LR=LS).

Turning now to Fig. 61B, a reciprocal square root base table index 470 is depicted. Similarly, Fig. 61C depicts a reciprocal square root difference table index 480. Both indices 470 and 480 are formed from the bit groups shown in Fig. 61A, and usable in a similar manner to indices 440 and 450 shown in Figs. 8C and 8D.

Turning now to Fig. 62, a block diagram of a multi-function bipartite look-up table 500 is shown according to one embodiment of the present invention. Look-up table 500 receives input value 420 (depicted above in Fig. 60A) and a function select signal 502, and generates an output value 550 as a result of the table look-up operation. Input value 420 and function select signal 502 are conveyed to an address control unit 510, which in turn generates a base table index 512 and a difference table index 514. Base table index 512 is conveyed to base table 520, which, in one embodiment, includes base output values for both the reciprocal function and the reciprocal square root function. Similarly, difference table index 514 is conveyed to difference table 530. Difference table 530 may also, in one embodiment, include difference output values for both the reciprocal and reciprocal square root functions.

In the embodiment shown in Fig. 62, base table 520 includes output base values for the reciprocal square root function over an input range of two binades. These base values are stored within locations in base table regions 522A and 522B. Table 520 further includes base output values for the reciprocal function over a single binade in entries within base table region 522C. Each region 522 includes a number of entries equal to the number of intervals in the allowable input range times the number of subintervals/interval.

Difference table 530, on the other hand, is configured similarly to base table 520, only it includes output difference values for the two functions. Like table 520, table 530 includes difference values over two binades for the reciprocal square root function (within entries in difference table regions 532A and 532B), and over a single binade for the reciprocal function (within entries in region 532C). Each of regions 532 includes a number of entries equal to the number of intervals in the input range times the number of sub-subintervals/subinterval.

Ultimately, base table index 512 and difference table index 514 select entries from base table 520 and difference table 530, respectively. The output of base table 520, base table output 524, is conveyed to an adder 540, which also receives difference table output 534, selected from difference table 530 by difference table index 514. Adder 540 also receives an optional rounding constant 542 as a third addend. If rounding is not needed, constant 542 is zero. Adder 540 adds quantities 524, 534, and 542, generating output value 550.

As described above, an efficient indexing implementation may be achieved by partitioning the input range identically for each function provided by look-up table 500. This allows the entries for both functions within tables 520 and 530 to each be addressed by a single index, even though each table includes values for two functions. In the embodiment shown in Fig. 62, the input range for the two functions (reciprocal and reciprocal square root) are partitioned such that a single index is generated per table portion. As will be shown in Fig. 63, the number of index bits is equal to the number of bits necessary to select a table region 522/532, plus the number of bits needed to select an entry within the chosen table region (the number of entries in each storage region for tables 520 and 530 is described above).

In one embodiment, each of the entries in base table 520 is P bits (P > 1). Each entry in difference table 530 is Q bits, where Q is less than P. As described above, the ratio of P to Q depends upon the slope of the function being represented. In general, where I is the number of intervals in a predetermined input range

and J is the number of subintervals/interval, Q is related to P by the relationship Q=P-(I+J)+c, where c is a constant which depends upon the slope of the function (specifically the largest slope in magnitude that occurs in the primary input interval).

For example, for the reciprocal function, c=1 since the maximum slope in interval [1,2) is 1 (at x=1). Similarly, for the reciprocal square root function, c=0, since the maximum slope in [1,4) is 0.5 (at x=1). Generally speaking, a function with a relatively high slope requires more bits in the difference entry to represent change from a corresponding base value. In one embodiment, for example, both the reciprocal and reciprocal square root functions have slopes which allow Q to be less than 0.5*P, while still maintaining a high degree of accuracy.

Adder 540 is configured to be an R-bit adder, where R is sufficient to represent the maximum value in base table 520 (R may be equal to P in one embodiment). Adder 540 is configured to add table outputs 524 and 534, plus optional rounding constant 542, such that the least significant bits of the addends are aligned. This add operation results in an output value 550 being produced. In one embodiment, the use of optional rounding constant 542 results in a number of least significant bits being discarded from output value 550.

In the embodiment shown in Fig. 62, adder 540 does not generate a carry out signal (a carry out signifies that output value 550 exceeds 2^R). Since all the entries of tables 520 and 530 have been determined before table 500 is to be used (during operation of a microprocessor in one embodiment), it may be determined if any of the possible combinations of base/difference entries (plus the rounding constant) result in an output value 550 which necessitates providing a carry out signal.

As shown, result 560 for the two functions of table 500 includes an output sign bit portion 562, an output exponent portion 564, and an output mantissa portion 566. Output value 550 is usable as mantissa portion 566, although some bits may be discarded from output value 550 in writing output mantissa portion 566. With regard to the value of output sign bit portion 562, the value of input sign portion 422 is usable as the value of portion 562 for both the reciprocal and reciprocal square root functions. The value of output exponent portion 564 is generated from the value of input exponent portion 422 of input value 420, and is calculated differently for the reciprocal function than it is for the reciprocal square root function.

In one embodiment, the true input exponent, TIEXPO, is related to the value of field 424 in input value 420, IEXPO. Similarly, the true output exponent, TOEXPO, is related to the value to be written to field 564, OEXPO. The value written to OEXPO is dependent upon the particular function being evaluated.

For the reciprocal function, the value written to OEXPO is computed such that TOEXPO=-1-TIEXPO[+CR], where [+CR] is part of the equation if carry out generation is applicable. For the common case in which IEXPO=TIEXPO+BIAS and OEXPO=TOEXPO+BIAS, it follows that OEXPO=2*BIAS-1-EXPO[+CR].

For the reciprocal square root function, OEXPO is computed such that TOEXPO=(-1-(TIEXPO/2))[+CR] if TIEXPO is greater than or equal to zero. Conversely, if TIEXPO is less than zero, OEXPO is computed such that TOEXPO=(-(TIEXPO+1/2))[+CR]. For the common case in which IEXPO=TIEXPO+BIAS and OEXPO=TOEXPO+BIAS, OEXPO=((3*BIAS-1-IEXPO)>>1)[+CR].

Turning now to Fig. 63, a block diagram of address control 510 within multi-function look-up table 500 is depicted according to one embodiment of the present invention. Address control unit 510 receives input value 420 and function select signal 502 and generates base table index 512 and difference table index 514.

Input value 420 includes sign bit field 422 having a value IS, exponent field 424 having a value IEXPO (the biased exponent value), and mantissa field 426 having a value IMANT. As shown, mantissa field 426 includes three bit groups (573, 574, and 575) usable to form indices 512 and 514. Because input value 420 is used to select base/difference values for both the reciprocal and reciprocal square root functions, these bit groups are equivalent to the bit groups of Figs. 8B and 9A. More specifically, group 573 is equivalent to groups 430 and 460, respectively, since group 573 is usable to specify an interval for both functions within table 500. Similarly, group 574 is equivalent to groups 432/462, while group 575 is equivalent to groups 434/464. Bit group 573 is shown as having XH bits, where XH=HR=HS. Similarly, bit group has XM bits (XM=MR=MS), while bit group 575 has XL bits (XL=LR=LS). Bit groups 573-575 are combined as shown in Figs. 8C-D (and 9B and 9C) in order to form portions of indices 512 and 514.

The most significant bits of indices 512 and 514 are used for function selection. In the embodiment shown in Fig. 63, the most significant bit is low when function select signal 502 is high (as signal 502 is conveyed through an inverter 570). Thus, when signal 502 is high, base table index 512 and difference table index 514 access entries within table regions 522A-B and 532A-B (the reciprocal square root entries). Conversely, when signal 502 is low, indices 512 and 514 access entries within table regions 522C and 532C (the reciprocal entries). The second most significant bit of indices 512/514 is used (if applicable) to select one of the two binades for the reciprocal square root entries. That is, these bits select between table regions 522A and 522B in base table 520, and between table regions 532A and 532B in difference table 530. Furthermore, these second-most-significant bits are only set (in the embodiment shown) if function select 502 is high and the LSB of the true exponent value is set (meaning the true exponent is odd and the biased exponent, 511, is even). Thus, these bits are not set if function select 502 is low, indicating the reciprocal function.

The equations for index 512 in the embodiment shown in Fig. 62 may be summarized as follows:

BADDR<XH+XM+1> =!(Signal 502), (11)
BADDR<XH+XM> =!IEXPO<0>&&(502), (12)
BADDR<XH+XM-1:XM> =IMANT<N-2:N-1-XH>, (13)
BADDR<XM-1:0> =IMANT<N-2-XH:N-1-XH-XM>. (14)

Similarly, the equations for index 514 are as follows:

DADDR<XH+XL+1> =!(Signal 502), (15)
DADDR<XH+XL> =IEXPO<0>&&(502), (16)
DADDR<XH+XL-1:XL> =IMANT<N-2:N-1-XH>, (17)
DADDR<XL-1:0> =IMANT<N-2-XH-XM:N-1-XH-XM-XR>. (18)

Other equations are possible in other embodiments.

Turning now to Fig. 64A, a graph 578 of an input region 580 is shown according to a prior art method for calculating a midpoint value. Input region 580 is bounded by input values A and B, located at points 582 and 584, respectively, on the horizontal axis of graph 578. Point A corresponds to an output value (for the reciprocal function) denoted by point 581 on the vertical axis of graph 578. Point B, likewise, corresponds to an output value denoted by point 583.

As shown in Fig. 64A, a midpoint X1 is calculated for input region 580 by determining the input value halfway in between A and B. This input value X1 is located at point 586, and corresponds to an output value denoted by point 585 on the vertical axis. In prior art systems, the output value corresponding to point 585 is chosen to represent all values in input region 580. An output value calculated in this manner has the effect of minimizing maximum relative error over a given input region. Although this midpoint calculation method is shown in Fig. 64A for the reciprocal function, this method is applicable to any function.

Turning now to Fig. 64B, a graph 590 of input region 580 is shown according to a method for calculating a midpoint value according to the present invention. As in Fig. 64A, input region 580 is bounded by input values A and B located at points 582 and 584, respectively. Input value A corresponds to an output value denoted by point 581, while input value B corresponds to an output value at point 583. As depicted in Fig. 64B, both of these output values correspond to the reciprocal function.

Unlike the midpoint calculation in Fig. 64A, the midpoint calculation in Fig. 64B produces an output value for input region 580 which minimizes absolute error. The midpoint calculation is Fig. 64A is independent of the particular function, since the midpoint (X1) is simply calculated to be halfway between the input values (A and B) which bound region 580. Midpoint X2, on the other hand, is calculated such that the corresponding output value, denoted by point 587, is halfway between the output values (581 and 583) corresponding to the input region boundaries. That is, the difference between 581 and 587 is equal to the difference between 587 and 583. The calculation of X2 (denoted by point 588) is function-specific. For the reciprocal function, X2 is calculated as follows:

$$\frac{1}{A}-\frac{1}{X2}=\frac{1}{M2}-\frac{1}{B} \text{ (19), or}$$

$$A\cdot X2\cdot B\left(\frac{1}{A}-\frac{1}{X2}=\frac{1}{X2}-\frac{1}{B}\right) \text{ (20),}$$

which simplifies to

$$X2\cdot B-A\cdot B=A\cdot B-A\cdot X2 \quad (21).$$

Solving for X2 gives

$$X2 = \frac{2 \cdot A \cdot B}{A + B}.$$

Calculating X2 for the reciprocal square root function gives

$$X2 = \frac{4 \cdot A \cdot B}{A + 2\sqrt{A \cdot B} + B}.$$

Calculation of midpoint X2 in this manner ensures that maximum absolute error is minimized by selecting f(X2) as the output value for input region 580. This is true because the absolute error at both A and B is identical with f(X2) selected as the output value for region 580.

Another method of calculating error, "ulp" (unit in last place) error, is currently favored by the scientific community. Generally speaking, ulp error is scaled absolute error where the scale factor changes with a) precision of the floating point number and b) the binade of a particular number. For example, for IEEE single-precision floating point format, 1 ulp for a number in binade [1,2) is 2^{-23}. The ulp method of midpoint calculation is utilized below in a method for computation of base and difference table values in one embodiment of the present invention.

Turning now to Fig. 65A, a flowchart of a method 600 for calculations of difference table entries is depicted according to one embodiment of the present invention. Method 600 is described with further reference to Fig. 65B, which is a graph 640 of a portion 642 of function f(x). Method 600 is described generally in relation to Fig. 65A, while Fig. 65B illustrates a particular instance of the use of method 600.

Method 600 first includes a step 602, in which the input range of f(x) is partitioned into I intervals, J subintervals/interval, and K sub-subintervals/subinterval. The partitioning choice directly affects the accuracy of the look-up table, as a more narrowly-partitioned input range generally leads to reduced output error. Fig. 65B illustrates a single interval 650 of the input range of f(x). Interval 650 is partitioned into four subintervals, 652A-D, each of which is further partitioned into four sub-subintervals. Subinterval 652A, for example, includes sub-subintervals 654A, 654B, 654C, and 654D.

These partitions affect the input regions for which difference table entries are generated. In one embodiment, a difference table entry is generated for each group of sub-subintervals in a subinterval of an input range. As described above, each sub-subinterval group includes one sub-subinterval/subinterval within a given interval, with each sub-subinterval in the group having the same relative position within its respective subinterval. For example, if an interval includes eight subintervals of eight sub-subintervals each, a difference table according to one embodiment of the present invention would include eight entries for the interval. Consider Fig. 65B. Interval 650 is shown as having four subintervals 652 of four sub-subintervals each. Each sub-subinterval within a given subinterval belongs to one of four groups. Each group has a number of entries equal to the number of subintervals/interval, and each member of a particular group has the same relative position within its respective subinterval. Group 2, for instance, includes sub-subintervals 654C, 655C, 656C,

and 657C, all of which are the third sub-subinterval within their respective subintervals. As will be described below, a difference table entry is computed for each group within a given interval.

In step 604, a particular interval M is selected for which to calculate K difference table entries. In Fig. 65B, interval M is interval 650. Method 600 is usable to calculate difference table entries for a single interval; however, the method may be applied repeatedly to calculate entries for each interval in an input range.

Next, in step 606, a group of K sub-subintervals (referred to in Fig. 65A as "Group N") are selected for which to calculate a difference entry. Typically, the groups are selected sequentially. For example, in Fig. 65B, group 0 (consisting of sub-subintervals 654A, 655A, 656A, and 657A) would typically be selected first.

In step 608, a counter variable, SUM, is reset. As will be described, this variable is used to compute an average of the difference values in each group. SUM is reset each time a new group of sub-subintervals is processed.

Step 610 includes several sub-steps which make up a single iteration in a loop for calculating a single difference entry. In sub-step 610A, a subinterval is selected in which to begin computation of the current difference table entry being calculated. The current subinterval is referred to as "P" within Fig. 65A. Subintervals are also typically selected in sequential order. For example, in calculating table entries for groups 0-3 in Fig. 65B, computations first begin in subinterval 652A, then subinterval 652B, etc.

In sub-step 610B, the midpoint (X1) and corresponding output value (R=f(X1)) are computed for the sub-subinterval of group N located within current subinterval P. For example, if the current subinterval P is 652A and the current group N is group 0, the midpoint and corresponding output value are computed for sub-subinterval 654A. In one embodiment, midpoint X1 is calculated as shown in Fig. 64B. That is, the midpoint X1 is calculated such that f(X1) is halfway between the maximum and minimum output values for the sub-subinterval for which the midpoint is being calculated. The midpoints (660A-660P) are shown in Fig. 65B for each sub-subinterval within interval 650.

Next, in sub-step 610C, a midpoint(X2) and corresponding output value (R2=f(X2)) are calculated for a reference sub-subinterval within current subinterval P. This reference sub-subinterval is the sub-subinterval within current subinterval P for which the base value is ultimately calculated (as is described below with reference to Fig. 66A). In one embodiment, the reference sub-subinterval is the last sub-subinterval within a given subinterval. In Fig. 65B, for example, the reference sub-subintervals are those in group 3.

In sub-step 610D, the difference between the midpoint output values (R1-R2) is added to the current value of SUM. This effectively keeps a running total of the difference values for the group being calculated. The difference values for each sub-subinterval are represented by vertical lines 662 in Fig. 65B. Note that the difference value for the reference sub-subinterval in each subinterval is zero.

In step 612, a determination is made whether current subinterval P is the last (J-1th) subinterval in interval M. If P is not the last subinterval in interval M, processing returns to step 610. In sub-step 610A, the next subinterval (sequential to that previously processed) is selected as subinterval P. Computations are made in sub-steps 610B-C of the midpoint and midpoint output values for the group N sub-subinterval and reference sub-subinterval within the newly-selected subinterval P. The new R1-R2 computation is performed and added to the SUM variable in sub-step 610D. This processing continues until all subintervals in interval M have been

traversed. For example, step 610 is executed four times to calculate a difference table entry for group 0 sub-subintervals in interval 650.

When step 612 is performed and current subinterval P is the last subinterval in interval M, method 600 continues with step 620. In step 620, the current value of SUM is divided by the number of times step 610 was performed (which is equal to the number of subintervals/intervals, or J). This operation produces a value AVG, which is indicative of the average of the difference values for a particular group. Entry 0 of the difference table for interval 650 corresponds to the sub-subintervals in group 0. This entry is calculated by the average of difference values represented by lines 662A, 662D, 662G, and 662J in Fig. 65B. Note that the difference entries for group 3 in this embodiment are zero since group 3 includes the reference sub-subintervals.

In step 622, the floating-point value AVG is converted to an integer format for storage in difference table 530. This may be performed, in one embodiment, by multiplying AVG by 2^{P+1}, where P is the number of bits in base table 520, and the additional bit accounts for the implicit leading one bit. A rounding constant may also be added to the product of AVG*2^{P+1} in one embodiment.

In step 624, the integer computed in step 622 may be stored to the difference table entry for interval M, sub-subinterval group N. Typically, all the entries for an entire table are computed during design of a microprocessor which includes table 500. The table values are then encoded as part of a ROM within the microprocessor during manufacture.

In step 630, a determination is made whether group N is the last sub-subinterval group in interval M. If group N is not the last group, method 600 continues with step 606, in which the next sub-subinterval group is selected. The SUM variable is reset in step 608, and difference table entry for the newly-selected sub-subinterval group is computed in steps 610, 612, 620, and 622. When group N is the last sub-subinterval group in interval M, method 600 completes with step 632. As stated above, method 600 is usable to calculate difference tables for a single interval. Method 600 may be repeatedly executed to calculate difference table entries for additional intervals of f(x).

As described above, the base value in look-up table 500 includes an approximate function value for each subinterval. As shown in Fig. 65B, this approximate function value for each subinterval corresponds to the midpoint of the reference sub-subinterval within the subinterval. For example, the approximate function value for subinterval 652A in Fig. 65B is the function value at midpoint 660D of sub-subinterval 654D. An approximate function value for another sub-subinterval within subinterval 652A may then be calculated by adding the function value at midpoint 660D with the difference table entry for the appropriate interval/sub-subinterval group.

Because of the averaging between subintervals used to compute difference table 530 entries, for a given interval (interval 650, for example), the differences (and, therefore, the result of the addition) are too small in the first subintervals in interval 650 (i.e., subintervals 652A-B). Conversely, the differences (and result of the addition) are too large in the last subintervals in interval 650 (subintervals 652C-D). Furthermore, within a given subinterval, error varies according to the sub-subinterval position due to difference value averaging. Difference value error from averaging refers to the difference between a computed midpoint for a sub-subinterval and the actual table output (a base-difference sum) for the group which includes the sub-subinterval. Within the last sub-subinterval in a subinterval, this error is zero. In the first sub-subinterval within the

subinterval, however, this error is at its maximum. In one embodiment, it is desirable to compute base table entries for a given subinterval such that maximum error is distributed evenly throughout the subinterval. Graphs illustrating the result of this process are depicted in Figs. 14A-D, with an actual method for this computation described with reference to Fig. 67.

Turning now to Fig. 66A, a graph 700 is shown of a portion of function f(x) (denoted by reference numeral 642) from Fig. 14B. Only subinterval 652A is shown in Fig. 66A. As in Fig. 65B, subinterval 652A includes four sub-subintervals (654A-D), each having a corresponding midpoint 660. Graph 700 further includes a line segment 702, which illustrates the actual look-up table outputs 704 for each sub-subinterval 654 of subinterval 652A.

These actual look-up table outputs are equal to the base entry plus the corresponding difference table entry. As described above, for the first subintervals (such as 652A) in subinterval 650, the result of the base-difference addition is smaller than computed midpoints for the sub-subintervals in the subinterval. This can be seen in Fig. 66A, as actual look-up table output 704A is less than computed midpoint 660A. Furthermore, for the embodiment shown in Fig. 66A, the sub-subinterval with the maximum error within subinterval 652A is sub-subinterval 654A. The difference between computed midpoint 660A and actual look-up table output 704A is shown as maximum error value 706. Actual look-up table outputs 704B and 704C in sub-subintervals 654B and 654C are also less than their respective computed midpoints, but not by as large a margin as in sub-subinterval 654A. Sub-subinterval 654D, however, is used as the reference sub-subinterval, and as a result, actual look-up table output 704D is equal to computed midpoint 660D.

Turning now to Fig. 66B, a graph 710 is shown of a portion of function f(x) (denoted by reference numeral 642) from Fig. 14B. Only subinterval 652D is shown in Fig. 66B. As in Fig. 65B, subinterval 652D includes four sub-subintervals (657A-D), each having a corresponding midpoint 660. Graph 710 further includes a line segment 712, which depicts the actual look-up table outputs 714 for each sub-subinterval 657 of subinterval 652D.

As in Fig. 66A, these actual look-up table outputs are equal to the base entry plus the corresponding difference table entry. As described above, for the last subintervals (such as 652D) in subinterval 650, the result of the base/difference addition is larger than computed midpoints for the sub-subintervals in the subinterval. This can be seen in Fig. 66B, as actual look-up table output 714A is greater than computed midpoint 660M. For the embodiment shown in Fig. 66B, the sub-subinterval with the maximum error is within subinterval 652D is sub-subinterval 657A. This difference between computed midpoint 660M and actual look-up table output 714A is shown as maximum error value 716. Actual look-up table outputs 714B and 714C in sub-subintervals 657B and 657C are also greater than their respective computed midpoints, but not by as large a margin as in sub-subinterval 657A. Sub-subinterval 657D, however, is used as the reference sub-subinterval, and as a result, actual look-up table output 714D is equal to computed midpoint 660P.

In one embodiment, the base value for a subinterval may be adjusted (from the function output value at the midpoint of the reference sub-subinterval) in order to more evenly distribute the maximum error value. Although adjusting the base values increases error within the reference sub-subinterval, overall error is evenly distributed across all sub-subintervals in a subinterval. This ensures that error is minimized within a subinterval no matter which sub-subinterval bounds the input value.

Turning now to Fig. 66C, a graph 720 is depicted which illustrates portion 642 of function f(x) corresponding to subinterval 652A. Graph 720 also includes a line segment 724, which is equivalent to line segment 702 with each table value adjusted by an offset. Values making up line segment 724 are adjusted such that the error in sub-subinterval 654A is equal to the error in sub-subinterval 654D. The error in sub-subinterval 654A is given by the difference between computed midpoint 660A of sub-subinterval 654A and adjusted look-up table output value 722A. This difference is denoted by -Δf(x) 726A in Fig. 66C. The error in sub-subinterval 654D is given by the difference between adjusted look-up table output value 722D and computed midpoint 660D of subinterval 654D. This difference is denoted by Δf(x) 726B. Thus, the error in sub-subinterval 654A and the error in sub-subinterval 654D are equal in magnitude, but opposite in sign.

Turning now to Fig. 66D, a graph 730 is depicted which illustrates portion 642 of function f(x) corresponding to subinterval 652D. Graph 730 also includes a line segment 734, which is equivalent to line segment 712 with each table value adjusted by an offset. Unlike the offset value in Fig. 66C, which is positive, the offset value in Fig. 66D is negative. With this offset value, the values which make up line segment 734 are adjusted such that the error in sub-subinterval 657A is equal to the error in sub-subinterval 657D. The error in sub-subinterval 657A is given by the difference between adjusted look-up table output value 732A and computed midpoint 660M. This difference is denoted by Δf(x) 736A in Fig. 66D. Similarly, the error in sub-subinterval 657D is given by the difference between computed midpoint 660P of subinterval 657D and adjusted look-up table output value 732D. This difference is denoted by -Δf(x) 736B. Thus, the error in sub-subinterval 657A and the error in sub-subinterval 657D are equal in magnitude, but opposite in sign. The method by which the adjustments of Figs. 14C and 14D are made is described below with reference to Fig. 67.

Turning now to Fig. 67, a flowchart of a method 800 is depicted for computing base table entries for a bipartite look-up table such as look-up table 500 of Fig. 62. Method 800 may be performed in conjunction with method 600 of Fig. 65A, or with other methods employed for computation of difference table entries. As needed, method 800 is also described with reference to Figs. 65A-D.

Method 800 first includes a step 802 in which the input range of f(x) is partitioned. Step 802 is identical to step 602 of method 600, since base and difference values are computed according to the same partitioning. Method 800 next includes step 804, in which difference table entries are calculated. This may be performed using method 600 or other alternate methods. In the embodiment shown in Fig. 67, difference entries are computed prior to base values since difference values are referenced during base value computation (as in step 822 described below).

Once difference table entries are calculated, computation of base table values begins with step 806, in which an interval (referred to as "M") is selected for which to calculate the entries. As with method 600, method 800 is usable to calculate entries for a single interval of a function input range. The steps of method 800 may be repeatedly performed for each interval in an input range. In the embodiment shown in Fig. 67, J base tables (one for each subinterval) are calculated for interval M. In step 810, one of the J subintervals of interval M is selected as a current subinterval P. The first time step 808 is performed during method 800, the first subinterval within interval M is selected as subinterval P. Successive subintervals are selected on successive executions of step 808. Currently selected subinterval P is the subinterval for which a base table entry is being calculated.

In step 810, an initial base value (B) is computed for currently selected subinterval P. In one embodiment, B corresponds to the function value at the midpoint (X2) of a predetermined reference sub-subinterval, where the midpoint is calculated as described with reference to Fig. 64B. (The midpoint of the reference sub-subinterval for subinterval P is denoted as X2 in order to be consistent with the terminology of Fig. 65A). The initial base value is thus given by the equation B=f(X2). In one embodiment of look-up table 500 (such as in Figs. 64B and 65A-D), the reference sub-subinterval (Q) is the last, or (K-1)th, sub-subinterval in each subinterval, where each subinterval includes sub-subintervals 0 to K-1.

Next, in step 812, a function value (D) is computed which corresponds to the midpoint (X3) of a sub-subinterval (R) within subinterval P which has the greatest difference value from reference sub-subinterval Q. If reference sub-subinterval Q is the last sub-subinterval in subinterval P, then sub-subinterval R is the first, or 0th, sub-subinterval. For example, in Fig. 66A, sub-subinterval 654D is reference sub-subinterval Q, while sub-subinterval 654A is sub-subinterval R. The function value D is thus given by the equation D=f(X3), where X3 is the midpoint of sub-subinterval R calculated as described above with reference to Fig. 64B in one embodiment.

In step 820, the difference, (referred to as "actual difference" in Fig. 67), is computed between D and B. This is representative of what the maximum difference value would be for subinterval P if difference value averaging were not employed, since sub-subinterval R has the maximum difference value in relation to sub-subinterval Q as described above. Next, in step 822, the difference table entry (computed previously in step 804) is referenced for subinterval P, sub-subinterval R. (In method 600, however, a dedicated difference table entry does not exist solely for subinterval P, sub-subinterval R. Rather, a difference table exists for subinterval P and a group of sub-subintervals N within interval M which includes sub-subinterval R). The difference table entry referenced in step 822 is referred to as the averaged difference value ("avg. diff.").

In step 824, the maximum error that results from using averaged difference values is calculated. This is performed by setting max error = actual diff. - avg. diff. As shown in Figs. 14C and 14D, the maximum error from the averaged difference table values occurs in the first sub-subinterval in the subinterval (e.g., sub-subintervals 654A and 657A). In fact, the max error computed in step 824 of method 800 is equal to max error values 706 and 716 in Figs. 14C and 14D.

In order to distribute the maximum error of step 824 throughout subinterval P, an adjust value is computed as a fraction of max error in step 826. In order to evenly distribute the error throughout the subinterval, the adjust value is computed as half the maximum error value. Then, in step 828, the final base value is computed from the initial base value B by adding the adjust value.

In step 830, the final value as computed in step 828 is converted to an integer value. As with the integer conversion of the difference value in step 622 of method 600, the conversion of step 830 may be performed in one embodiment by multiplying the final base value by 2^{P+1} and adding an optional rounding constant. In alternate embodiments, the integer conversion may be performed differently. In step 832, the converted integer value is ready for storage to the base table entry for interval M, subinterval P. The base table entries may be stored to the table one-by-one, but typically they are all computed then stored to the ROM that includes the look-up table.

In step 834, a determination is made of whether subinterval P is the last subinterval in interval M. If more subintervals exist, method 800 continues with step 808. In step 808, a next subinterval within interval M is selected, and the succeeding steps are usable to calculate the base value for the newly-selected subinterval. On the other hand, if P is the last subinterval in interval M, method 800 concludes with step 836.

Methods for calculation of difference and base table entries are described in a general manner with reference to Figs. 13A and 15, respectively. Source code which implements these methods (for the reciprocal and reciprocal square root functions) is shown below for one embodiment of the present invention. Note that the #define's for HIGH, MID, and LOW effectively partition the input range of these functions into four intervals, four subintervals/interval, and four sub-subintervals/subinterval.

```
#define HIGH        2
#define MID         2
#define LOW         2
#define OUT         16
#define OUTP        16
#define OUTQ        (OUTP-(HIGH+MID)+1)
#define RECIPENTRIES  (1L << (HIGH+MID))
#define ROOTENTRIES   (2L << (HIGH+MID))

#define BIAS 127L          /* exponent bias for single precision format */
#define POW2(x) (1L << (x))  /* helper function */


typedef union {
  float        f;
  unsigned long  i;
} SINGLE;

#define SIGN_SINGLE(var) ((((var).i)&0x80000000L)?1L:0L)      /* sign bit */
#define EXPO_SINGLE(var) ((((var).i)>>23L)&0xFFL)       /* 8 bit exponent */
#define MANT_SINGLE(var) (((var).i)&0x7FFFFFL)        /* 23 bit mantissa */

#define SETSIGN_SINGLE(var,sign) \
(((var).i)=((sign)&1)?(((var).i)|0x80000000L):(((var).i)&0x7FFFFFFFL))

#define SETEXPO_SINGLE(var,expo) \
(((var).i)=(((var).i)&0x807FFFFFL)|(((expo)&0xFFL)<<23))

#define SETMANT_SINGLE(var,mant) \
(((var).i)=(((var).i)&0xFF800000L)|(((mant)&0x7FFFFFL)))

extern unsigned long rom_p[];
extern unsigned long rom_q[];

#define TRUE  1
#define FALSE 0
#define HIGHMID (HIGH+MID)
#define HIGHLOW (HIGH+LOW)
#define ALL    (HIGH+MID+LOW)
#define POW2(x) (1L << (x))
#define CONCAT(a,b,c)  ((0x7FL << 23) | \
              (((a) & (POW2(HIGH) - 1)) << (23 - (HIGH))) | \
              (((b) & (POW2(MID)  - 1)) << (23 - (HIGHMID))) | \
```

```
                (((c) & (POW2(LOW) - 1)) << (23 - (ALL))))

#define CONCAT2(e,a,b,c) (((e) << 23) | \
                (((a) & (POW2(HIGH) - 1)) << (23 - (HIGH))) | \
                (((b) & (POW2(MID) - 1)) << (23 - (HIGHMID))) | \
                (((c) & (POW2(LOW) - 1)) << (23 - (ALL))))


void make_recip_bipartite_table (unsigned long *tablep, unsigned long *tableq)
{
  unsigned long xh, xm, xl, indexp, indexq, maxq, minq, maxp, minp;
  SINGLE      temp1, temp2;
  double      midpoint1, midpoint2;
  double      result, sumdiff, result1, result2, adjust ;

  printf ("\nCreating lookup tables ...\n");

  for (xh = 0; xh < POW2(HIGH); xh++) {
    for (xl = 0; xl < POW2(LOW); xl++) {
      indexq = (xh << LOW) | xl;
      sumdiff = 0.0;
      for (xm = 0; xm < POW2(MID); xm++) {
        temp1.i   = CONCAT (xh, xm, xl);
        temp2.i   = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
        midpoint1 = (2.0 * temp1.f * temp2.f) / (temp1.f + temp2.f);

        temp1.i   = CONCAT (xh, xm, POW2(LOW)-1);
        temp2.i   = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
        midpoint2 = (2.0 * temp1.f * temp2.f) / (temp1.f + temp2.f);

        sumdiff   = sumdiff + ((1.0 / midpoint1) - (1.0 / midpoint2));
      }
      result = 1.0/((double)(POW2(MID))) * sumdiff;
      tableq [indexq] = (unsigned long)(POW2(OUTP+1) * result + 0.5);
    }
  }

  for (xh = 0; xh < POW2(HIGH); xh++) {
    for (xm = 0; xm < POW2(MID); xm++) {
      indexp = (xh << (MID)) | xm;
      temp1.i   = CONCAT (xh, xm, 0);
      temp2.i   = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
      midpoint1 = (2.0 * temp1.f * temp2.f) / (temp1.f + temp2.f);
      result1   = 1.0 / midpoint1;

      temp1.i   = CONCAT (xh, xm, POW2(LOW) - 1);
      temp2.i   = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
      midpoint2 = (2.0 * temp1.f * temp2.f) / (temp1.f + temp2.f);
      result2   = 1.0 / midpoint2;

      adjust = 0.5 * ((result1 - result2) - (1.0/POW2(OUTP+1)) * tableq[xh << LOW]);

      tablep [indexp] = (unsigned long)(POW2(OUTP+1) * (result2 + adjust) + 0.5);
      tablep [indexp] -= (1L << OUTP);  /* subtract out integer bit */
    }
  }
}
```

```
void make_recipsqrt_bipartite_table (unsigned long *tablep,
                    unsigned long *tableq)
{
  unsigned long xh, xm, xl, indexp, indexq, maxq, minq, start, end,
          maxp, minp, expo;
  SINGLE      temp1, temp2;
  double      midpoint1, midpoint2;
  double      result, adjust, sumdiff, result1, result2;

  printf ("\nCreating lookup tables ...\n");
  for (expo = 0x7F; expo <= 0x80; expo++) {
    for (xh = 0; xh < POW2(HIGH); xh++) {
      for (xl = 0; xl < POW2(LOW); xl++) {
        indexq = ((expo & 1) << (HIGHLOW)) | (xh << LOW) | xl;
        sumdiff = 0.0;
        for (xm = 0; xm < POW2(MID); xm++) {
          temp1.i  = CONCAT2 (expo, xh, xm, xl);
          temp2.i  = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
          midpoint1 = (4.0 * temp1.f * temp2.f) / ((sqrt(temp1.f)+sqrt(temp2.f))*(sqrt(temp1.f)+sqrt(temp2.f)));

          temp1.i  = CONCAT2 (expo, xh, xm, POW2(LOW)-1);
          temp2.i  = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
          midpoint2 = (4.0 * temp1.f * temp2.f) / ((sqrt(temp1.f)+sqrt(temp2.f))*(sqrt(temp1.f)+sqrt(temp2.f)));

          sumdiff  = sumdiff + ((1.0 / sqrt(midpoint1)) - (1.0 / sqrt(midpoint2)));
        }
        result = 1.0/((double)(POW2(MID))) * sumdiff;
        tableq [indexq] = (unsigned long)(POW2(OUTP+1) * result + 0.5);
      }
    }

    for (xh = 0; xh < POW2(HIGH); xh++) {
      for (xm = 0; xm < POW2(MID); xm++) {
        indexp  = ((expo & 1) << (HIGHMID)) | (xh << (MID)) | xm;
        temp1.i  = CONCAT2 (expo, xh, xm, 0);
        temp2.i  = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
        midpoint1 = (4.0 * temp1.f * temp2.f) / ((sqrt(temp1.f)+sqrt(temp2.f))*(sqrt(temp1.f)+sqrt(temp2.f)));
        result1  = 1.0 / sqrt(midpoint1);

        temp1.i  = CONCAT2 (expo, xh, xm, POW2(LOW) - 1);
        temp2.i  = (temp1.i | (POW2(23 - ALL) - 1)) + 1;
        midpoint2 = (4.0 * temp1.f * temp2.f) / ((sqrt(temp1.f)+sqrt(temp2.f))*(sqrt(temp1.f)+sqrt(temp2.f)));
        result2  = 1.0 / sqrt(midpoint2);

        adjust = 0.5 * ((result1 - result2) - (1.0/POW2(OUTP+1)) * tableq[((expo & 1) << (HIGH+LOW)) | (xh
  << LOW)]);

        tablep [indexp] = (unsigned long)(POW2(OUTP+1) * (result2 + adjust) + 0.5);
        tablep [indexp] -= (1L << OUTP); /* subtract out integer bit */
      }
    }
  }
}


void recip_approx_bipartite (
  const SINGLE *arg,
  const unsigned long *tablep,
```

```
const unsigned long *tableq,
unsigned long  high,
unsigned long  mid,
unsigned long  low,
unsigned long  out,
SINGLE *approx)
{
  unsigned long  expo, sign, mant, indexq, indexp, p, q;

  /* handle zero separately */

  if ((arg->i & 0x7F800000L) == 0) {
    approx->i = (arg->i & 0x80000000L) | 0x7F7FFFFFL;
    return;
  }

  /* unpack arg */

  expo  = (arg->i >> 23) & 0xFF;
  sign  = (arg->i >> 31) & 1;
  mant  = (arg->i & 0x7FFFFFL);

  /* do table lookup on tables P and Q */

  indexp = (mant >> (23 - (high + mid)));
  indexq = ((mant >> (23 - (high))) << low) |
        ((mant >> (23 - (high+mid+low))) & (POW2(low) - 1));
  p = tablep [indexp];
  q = tableq [indexq];

  /* generate result in single precision format */

  approx->i = ((2*BIAS + ~expo) << 23L) +
          (((p + q)) << (23L - out));

  /* check for underflow */

  if ((((approx->i >> 23) & 0xFFL) == 0x00L) ||
     (((approx->i >> 23) & 0xFFL) == 0xFFL)) {
    approx->i = 0L;
  }

  /* mask sign bit because exponent above may have overflowed into sign bit */

  approx->i = (approx->i & 0x7FFFFFFFL) | (sign << 31L);
}


void recipsqrt_approx_bipartite (
  const SINGLE *arg,
  const unsigned long *tablea,
  const unsigned long *tableb,
  unsigned long  high,
  unsigned long  mid,
  unsigned long  low,
  unsigned long  out,
  SINGLE *approx)
```

```
{
unsigned long sign, mant, indexq, indexp, p, q;
long expo;

/* Handle zero separately. Returns maximum normal */

if ((arg->i & 0x7F800000L) == 0L) {
  approx->i = 0x7F7FFFFFL | (arg->i & 0x80000000L);
  return;
}

expo  = (arg->i >> 23) & 0xFFL;
sign  = (arg->i >> 31) & 1;
mant  = (arg->i & 0x7FFFFFL);
indexp = ((expo & 1) << (high + mid)) | (mant >> (23 - (high + mid)));
indexq = ((expo & 1) << (high + low)) | ((mant >> (23 - (high))) << low) |
      ((mant >> (23 - (high + mid + low))) & (POW2(low) - 1));
p = tablea [indexp];
q = tableb [indexq];

approx->i = (((3*BIAS + ~expo) >> 1) << 23) +
        (((p + q)) << (23 - out));

approx->i |= sign << 31;
}
```

To further clarify calculation of base and difference table entries in the embodiment corresponding to the above source code, sample table portions are given below. These table portions are for the reciprocal function only, although the reciprocal square root table entries are calculated similarly. The input range (1.0 inclusive to 2.0 exclusive) for this example is divided into four intervals, four subintervals/interval, and four sub-subintervals/subinterval. The table values are only shown for the first interval (1.0 inclusive to 1.25 exclusive) for simplicity.

The difference table for this example receives a four bit index (two bits for the interval, two bits for the sub-subinterval group). The base table also receives a four bit index (two bits for the interval, two bits for the subinterval). The base table includes 16 bits, while the difference table includes 13 bits for this embodiment.

Int.	Sub int.	Sub-Sub.	A	B	A (Binary)
0	0	0	1.0	1.015625	1.00 00 00 ...
0	0	1	1.015625	1.03125	1.00 00 01 ...
0	0	2	1.03125	1.046875	1.00 00 10 ...
0	0	3	1.046875	1.0625	1.00 00 11 ...
0	1	0	1.0625	1.078125	1.00 01 00 ...
0	1	1	1.078125	1.093125	1.00 01 01 ...
0	1	2	1.093125	1.109375	1.00 01 10 ...
0	1	3	1.109375	1.125	1.00 01 11 ...
0	2	0	1.125	1.140625	1.00 10 00 ...
0	2	1	1.140625	1.15625	1.00 10 01 ...
0	2	2	1.15625	1.171875	1.00 10 10 ...
0	2	3	1.171875	1.1875	1.00 10 11 ...
0	3	0	1.1875	1.203125	1.00 11 00 ...
0	3	1	1.203125	1.21875	1.00 11 01 ...
0	3	2	1.21875	1.234375	1.00 11 10 ...
0	3	3	1.234375	1.25	1.00 11 11 ...

Table 3

Table 1 illustrates the partitioning of the first interval of the input range of the reciprocal function. With regard to the binary representation of A, only the six high-order mantissa bits are shown since these are the ones that are used to specify the interval, subinterval, and sub-subinterval group of the input sub-region. Note that the first group of mantissa bits of A corresponds to the interval number, the second group corresponds to the subinterval number, and the third group corresponds to the sub-subinterval group.

Table 2 shows the midpoint of each sub-subinterval (computed as in Fig. 54B), as well as the function evaluation at the midpoint and the difference value with respect to the reference sub-subinterval of the subinterval. (The reference sub-subintervals are those in group 3).

Subint.	Sub-Sub.	Midpoint (M)	f(M)=1/M	Diff. Value
0	0	1.007751938	.992307692	.04410751672
0	1	1.023377863	.977156177	.02895600156
0	2	1.039003759	.962460426	.01426024955
0	3	1.05462963	.948200175	0
1	0	1.070255474	.934356352	.03920768144
1	1	1.085881295	.920910973	.02576230329
1	2	1.101507092	.907847083	.01269841270
1	3	1.117132867	.895148670	0
2	0	1.132758621	.882800609	.03508131058
2	1	1.148384354	.870788597	.02306929857
2	2	1.164010067	.859099099	.01137980085
2	3	1.179635762	.847719298	0
3	0	1.195261438	.836637047	.03157375602
3	1	1.210887097	.825840826	.0207775347
3	2	1.226512739	.815319701	.01025641026
3	3	1.242138365	.805063291	0

Table 4

Table 3 shows the difference value average for each sub-subinterval group. Additionally, Table 3 includes the difference average value in integer form. This integer value is calculated by multiplying the difference average by 2^{17}, where 17 is the number of bits in the output value (including the leading one bit).

Sub-Sub. Group	Difference Average	Integer Value (hex)
0	.03749256619	1332
1	.02464128453	0C9E
2	.01214871834	0638
3	0	0000

Table 5

With regard to the base values for this example, Table 4 below shows midpoints X2 and X3. Midpoint X2 is the midpoint for the reference sub-subinterval of each subinterval, while X3 is the midpoint of the sub-subinterval within each subinterval that is furthest from the reference sub-subinterval. The table also shows the function values at these midpoints.

Subint.	Midpoint X2	Init. Base Value (1/X2)	Midpoint X3	1/X3
0	1.05462963	.9482001756	1.007751938	.992307692
1	1.117132867	.8951486698	1.070255474	.934356352
2	1.179635762	.8477192982	1.132758621	.882800609
3	1.242138365	.8050632911	1.195261438	.836637047

Table 6

Next, Table 5 below shows the actual error difference for each subinterval, computed as 1/X3-1/X2. Table 5 additionally shows the average difference value, which is equal to the previously computed difference value for sub-subinterval group 0. The difference between the actual difference and the average difference is equal to the maximum error for the subinterval. Half of this value is the adjust value.

Subint.	Actual Diff. (1/X3-1/X2)	Average Diff.	Maximum Error	Adjust Value
0	.044107516	.03749256619	.00661495	.003307475
1	.039207682	.03749256619	.001715116	.000857558
2	.0358081311	.03749256619	-.002411255	-.001205628
3	.031573756	.03749256619	-.00591881	-.002959405

Table 7

In Table 6, The adjust value plus the initial base value gives the final base value. This final base value is converted to an 16-bit integer value by multiplying by 2^{17} and discarding the most significant 1 bit (which corresponds to the integer position).

Subint.	Final Base Value	Integer Value (hex)
0	.951507651	E72C
1	.896006228	CAC1
2	.846513671	B16A
3	.802103886	9AAD

Table 8

As stated above, the bipartite table look-up operation is usable to obtain a starting approximation for mathematical functions such as the reciprocal and reciprocal square root implemented within a microprocessor.

In one embodiment, the table look-up is initiated by a dedicated instruction within the instruction set of the microprocessor. Additional dedicated instructions may be employed in order to implement the iterative evaluations which use the starting approximation to produce the final result for these functions. This, in turn, leads to a faster function evaluation time.

In one embodiment, base and difference values calculated as described in Figs. 13A and 15 result in table output values with minimized absolute error. Advantageously, this minimal absolute error is obtained with a bipartite table configuration, which requires less table storage than a naive table of comparable accuracy. This configuration also allows the interpolation to be achieved with a simple addition. Thus, a costly multiply or multiply-add is not required to generate the final table output, effectively increasing the performance of the table look-up operation.

It is noted that while base and difference tables have been described above with reference to the reciprocal and reciprocal square root functions, such tables are generally applicable to any monotonically decreasing function. These tables are also applicable to a function which is monotonically decreasing over the desired input range.

In another embodiment, these base and difference tables may be modified to accommodate any monotonically increasing function(such as sqrt(x)), as well as any function which is monotonically increasing over a desired input range. In such an embodiment, the "leftmost" sub-subinterval within an interval becomes the reference point instead of the "rightmost" sub-subinterval, ensuring the values in the difference tables are positive. Alternatively, the "rightmost" sub-subinterval may still be used as the reference point if difference values are considered negative and a subtractor is used to combine base and difference table values.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

WHAT IS CLAIMED IS:

1. A method for generating entries for a bipartite look-up table having a base table portion and a difference table portion, wherein said bipartite look-up table is usable to generate output values for a given mathematical function over a predetermined input range which is divided into a plurality of intervals, said method comprising:

computing a first base table entry which corresponds to a first sub-range within a first interval of said predetermined input range, wherein said first interval includes a plurality of sub-ranges each divided into a plurality of sub-sub-ranges;

computing a first difference table entry which corresponds to a first group of sub-sub-ranges within said first interval, wherein said first group of sub-sub-ranges includes a first sub-sub-range which is located within said first sub-range;

wherein said first base table entry and said first difference table entry are usable to generate a first output value for an input value located within said first sub-sub-range, wherein said first output value has a minimized amount of absolute error for all input values within said first sub-sub-range.

2. A bipartite look-up table, comprising:

a base table portion for storing base entries for a given mathematical function over a predetermined input range, wherein said predetermined input range is divided into a plurality of intervals, wherein said base table portion includes a first base table entry corresponding to a first sub-range within a first interval of said predetermined input range;

a difference table portion for storing difference entries for said given mathematical function over said predetermined input range, wherein said difference table portion includes a first difference table entry corresponding to a first group of sub-sub-ranges within said first interval, wherein said first group of sub-sub-ranges includes a first sub-sub-range which is located within said first sub-range;

wherein said bipartite look-up table is configured to generate a first output value in response to receiving a first input value located within said first sub-sub-range, wherein said first output value has a minimized amount of absolute error for all input values within said first sub-sub-range.

3. A method for generating entries for a bipartite look-up table which includes a base table and a difference table, wherein said bipartite look-up table is configured to provide an output value for a given mathematical function in response to receiving a corresponding input value within a predetermined input range, said method comprising:

(a) dividing said predetermined input range into a predetermined number of equal intervals including a first interval;

(b) dividing said first interval into a predetermined number of equal subintervals including a first subinterval, wherein said first subinterval includes said corresponding input value;

(c) dividing each of said predetermined number of equal subintervals in said first interval into a predetermined number of equal sub-subintervals including a first sub-subinterval within said first subinterval, wherein said first sub-subinterval includes said corresponding input value;

(d) computing a first difference table entry which corresponds to a given group of sub-subintervals within said first interval, wherein each of said given group of sub-subintervals is located within a corresponding one of said predetermined number of equal subintervals within said first interval, and wherein each of said given group of sub-subintervals has a common relative position within said corresponding one of said predetermined number of equal subintervals within said first interval, wherein said given group of sub-subintervals includes said first sub-subinterval;

(e) computing a first base table entry which corresponds to said first subinterval within said first interval;

wherein said first difference table entry and said first base table entry are usable to generate said output value of given mathematical function for said corresponding input value, and wherein said first difference table entry and said first base table entry are computed such that said output value has a minimum amount of possible absolute error for all input values within said first sub-subinterval within said first subinterval of said first interval.

4. The method of claim 3, wherein said computing said first difference table entry includes:

(i) selecting a given subinterval within said first interval as a currently selected subinterval;

(ii) calculating a first midpoint value for a particular sub-subinterval within said currently selected subinterval which is included in said given group of sub-subintervals, wherein evaluating said given mathematical function at said first midpoint value produces a first function value which minimizes absolute error for all input values within said particular sub-subinterval;

(iii) calculating a second midpoint value for a reference sub-subinterval of said currently selected subinterval, wherein evaluating said given mathematical function at said second midpoint value produces a second function value which minimizes absolute error for all input values within said reference sub-subinterval of said currently selected subinterval;

(iv) computing a difference value between said first function value and said second function value;

(v) repeating steps (i)-(iv) using each remaining subinterval in said first interval as said currently selected subinterval, wherein said repeating includes computing a difference total of each said difference value;

(vi) determining an difference value average from said difference total;

(vii) storing said difference value average as said first difference table entry;

wherein said first difference table entry is usable to compute a value of said given mathematical function for an input value located in any of said given group of sub-subintervals within said first interval.

5. The method of claim 4 wherein said particular sub-subinterval of said currently selected subinterval for which said first midpoint value is calculated includes a first input range having a smallest first input value and a largest first input value, wherein evaluating said given mathematical function at said smallest first input value produces a third function value, and wherein evaluating said given mathematical function at said greatest first input value produces a fourth function value.

6. The method of claim 3, further comprising dividing each remaining one of said predetermined number of equal intervals into said predetermined number of equal subintervals, and dividing each of said predetermined number of equal subintervals within said each remaining one of said predetermined number of equal intervals into said predetermined number of equal sub-subintervals.

7. A method for generating entries for a difference table portion of a bipartite look-up table including difference values of a given mathematical function over a predetermined input range, said method comprising:

(a) dividing said predetermined input range into a predetermined number of equal intervals including a first interval;

(b) dividing said first interval into a predetermined number of equal subintervals including a first subinterval;

(c) dividing each of said predetermined number of equal subintervals in said first interval into a predetermined number of equal sub-subintervals, wherein each of said predetermined number of equal subintervals in said first interval includes a reference sub-subinterval;

(d) selecting a first group of sub-subintervals within said first interval, wherein each of said first group of sub-subintervals is located in a corresponding subinterval of said first interval, and wherein each of said first group of sub-subintervals has a common relative position with said corresponding subinterval;

(e) calculating a first difference table entry for said first group of sub-subintervals, wherein said first difference table entry is an average of difference values computed between each of said first group of sub-subintervals and said reference sub-subintervals in said corresponding subinterval;

and wherein said first difference table entry is usable to compute a function output value which has minimal absolute error for all possible input values within said first group of sub-subintervals.

8. The method of claim 7, wherein said calculating said first difference table entry includes:

(i) selecting a given subinterval as a currently selected subinterval of said first interval;

(ii) calculating a first midpoint value for a particular sub-subinterval of said currently selected subinterval which is included in said first group of sub-subintervals, wherein evaluating said given mathematical function at said first midpoint value produces a first function value which minimizes absolute error for all input values within said particular sub-subinterval of said currently selected subinterval;

(iii) calculating a second midpoint value within said reference sub-subinterval of said currently selected subinterval, wherein evaluating said given mathematical function at said second midpoint value produces a second function value which minimizes absolute error for all input values within said reference sub-subinterval of said currently selected subinterval;

(iv) computing a difference value between said first function value and said second function value;

(v) repeating steps (i)-(iv) using each remaining subinterval in said first interval as said currently selected subinterval, wherein said repeating includes summing each said difference value to produce a difference total;

(vi) determining an difference value average from said difference total;

(vii) storing said difference value average as said first difference table entry;

wherein said first difference entry is usable to compute a value of said given mathematical function for an input value located of any of said first group of sub-subintervals.

9. The method of claim 8, wherein said given mathematical function is f(x)=1/x or f(x)=1/sqrt(x).

10. A microprocessor, comprising:

an execution unit coupled to receive a first set of input data values, a second set of input data values, and an instruction indication specifying an operation to be performed by said execution unit, wherein said execution unit includes:

an input multiplexer unit coupled to receive said first set of input data values, said second set of input data values, and said instruction indication, wherein said input multiplexer is configured to select a first set of operands and a second set of operands from said first pair of input data values and said second pair of input data values in response to said instruction indication;

a first add/subtract pipeline coupled to receive said first set of operands and said instruction indication, wherein said first add/subtract pipeline is configured to generate a first result value from said first set of operands by performing an arithmetic operation specified by said instruction indication;

a second add/subtract pipeline coupled to receive said second set of operands and said instruction indication, wherein said second add/subtract pipeline is configured to generate a second result value from said second set of operands by performing said arithmetic operation specified by said instruction indication;

wherein said first result value and said second result value are generated concurrently.

11. The microprocessor of claim 10, wherein said first pair of input data values includes a first data value and a second data value, and wherein said second pair of input data values includes a third data value and a fourth data value, wherein said input multiplexer unit is configured to select said first set of operands to include said first data value and said third data value in response to said instruction indication specifying a vectored add operation, and wherein said input multiplexer unit is further configured to select said second set of operands to include said second data value and said fourth data value in response to said instruction indication specifying said vectored add operation.

12. The microprocessor of claim 11, wherein said first add/subtract pipeline is configured to generate said first result value from a sum of said first data value and said third data value in response to receiving said first data value, said third data value, and said instruction indication specifying said vectored add operation,

and wherein said second add/subtract pipeline is configured to generate said second result value from a sum of said second data value and said fourth data value in response to receiving said second data value, said fourth data value, and said instruction indication specifying said vectored add operation.

13. The microprocessor of claim 12, wherein said input multiplexer unit is configured to select said first set of operands to include said first data value and said second data value in response to said instruction indication specifying a vectored accumulate operation, and wherein said input multiplexer unit is furthered configured to select said second set of operands to include said third data value and said fourth data value in response to said instruction indication specifying said vectored accumulate operation.

14. The microprocessor of claim 11, wherein said input multiplexer unit is configured to select said first set of operands to include said first data value and said third data value in response to said instruction indication specifying either a vectored subtract operation or a vectored reverse subtract operation, and wherein said input multiplexer unit is furthered configured to select said second set of operands to include said second data value and said fourth data value in response to said instruction indication specifying either said vectored subtract operation or said vectored reverse subtract operation.

15. The microprocessor of claim 10, wherein said first pair of input data values includes a first floating point number, and wherein said second pair of input data values includes a second floating point number, wherein said input multiplexer unit is configured to select said first set of operands to include said first floating point number in response to said instruction indication specifying a floating point to integer conversion operation, and wherein said input multiplexer unit is configured to select said second set of operands to include said second floating point number in response to said instruction indication specifying said floating point-to-integer conversion operation.

16. The microprocessor of claim 15, wherein said first add/subtract pipeline is configured to generate said first result value by converting said first floating point number into a corresponding first integer value in response to receiving said first floating point number and said instruction indication specifying said floating point-to-integer conversion operation, and wherein said second add/subtract pipeline is configured to generate said second result value by converting said second floating point number into a corresponding second integer value in response to receiving said second floating point number and said instruction indication specifying said floating point-to-integer conversion operation.

17. The microprocessor of claim 11, wherein said first add/subtract pipeline includes a first far data path and a first close data path, and wherein said second add/subtract pipeline includes a second far data path and a second close data path, wherein said first far data path and said second far data path are configured to perform effective addition operations on received operands, and wherein said first far data path and said second far data path are further configured to perform effective subtraction operations on pairs of received floating point operands having an absolute exponent difference greater than one.

18. A microprocessor, comprising:

an execution unit coupled to receive a first set of input data values, a second set of input data values, and an instruction indication specifying an operation to be performed by said execution unit, wherein said execution unit includes:

a first add/subtract pipeline coupled to receive a first set of operands and said instruction indication, wherein said first set of operands are selected from said first set of input data values and said second set of input data values, and wherein said first add/subtract pipeline is configured to generate a first result value from said first set of operands according to said instruction indication;

a second add/subtract pipeline coupled to receive a second set of operands and said instruction indication, wherein said second set of operands are selected from said first set of input data values and said second set of input data values, and wherein said second add/subtract pipeline is configured to generate a second result value from said second set of operands according to said instruction indication;

an output multiplexer unit coupled to receive said first result value, said second result value, said instruction indication, and one or more additional input values, wherein said output multiplexer unit is configured to select a first output value and a second output value from said first result value, said second result value, and said one or more additional input values according to said instruction indication;

wherein said first result value and said second result value are generated concurrently.

19. The microprocessor of claim 18, wherein said first result value and said second result value are generated by an operation selected from the group consisting of: (i) vectored add operation, (ii) vectored subtract operation, (iii) vectored accumulate operation, (iv) vectored reverse subtract operation, (v) floating point-to-integer conversion operation, and (vi) integer-to-floating point conversion operation.

20. The microprocessor of claim 19, wherein said first result value and said second result value correspond to one of a plurality of arithmetic operations executable by said first add/subtract pipeline and said second add/subtract pipeline, wherein said first add/subtract pipeline includes a first far data path and a first close data path, and wherein said second add/subtract pipeline includes a second far data path and a second close data path.

21. The microprocessor of claim 20, wherein said first far data path and said second far data path are configured to perform effective addition operations on received operands, and wherein said first far data path and said second far data path are further configured to perform effective subtraction operations on pairs of received floating point operands having an absolute exponent difference greater than one.

22. The microprocessor of claim 21, wherein said first close data path and said second close data path are configured to perform effective subtraction operations on pairs of received floating point operands having an absolute exponent difference less than or equal to one.

23. A microprocessor, comprising:

an execution unit coupled to receive a first set of input data values, a second set of input data values, and an instruction indication specifying an operation to be performed by said execution unit, wherein said execution unit includes:

an input multiplexer unit coupled to receive said first set of input data values, said second set of input data values, and said instruction indication, wherein said input multiplexer is configured to select a first set of operands and a second set of operands from said first pair of input data values and said second pair of input data values according to said instruction indication;

a first add/subtract pipeline coupled to receive said first set of operands and said instruction indication, wherein said first add/subtract pipeline is configured to generate a first result value from said first set of operands according to said instruction indication;

a second add/subtract pipeline coupled to receive said second set of operands and said instruction indication, wherein said second add/subtract pipeline is configured to generate a second result value from said second set of operands according to said instruction indication;

an output multiplexer unit coupled to receive said first result value, said second result value, said instruction indication, and one or more additional input values, wherein said output multiplexer unit is configured to select a first output value and a second output value from said first result value, said second result value, and said one or more additional input values according to said instruction indication.

24. The microprocessor of claim 23, wherein said input multiplexer is configured to selectively route data values in said first set of input data values and said second set of input data values to said first add/subtract pipeline and said second add/subtract pipeline in response to said instruction indication specifying one of a first plurality of arithmetic operations.

25. A microprocessor, comprising:

an execution unit coupled to receive a first pair of floating point input values and a first control value indicative of an operation to be performed on said first pair of floating point input values, wherein said first pair of floating point input values includes a first floating point number and a second floating point number, wherein said execution unit includes:

a far data path coupled to receive said first pair of floating point input values and said first control value;

a close data path coupled to concurrently receive said first pair of floating point input values and said first control value;

wherein an effective addition operation is performed on said first pair of floating point input values in said far data path in response to said first control value indicating said effective addition operation;

wherein an effective subtraction operation is performed on said first pair of floating point input values in response to said first control value indicating said effective subtraction operation, wherein said effective subtraction operation is performed in said far data path in response to an absolute exponent difference of said first pair of floating point numbers being greater than one, and wherein said effective subtraction operation is

performed in said close data path in response to said absolute exponent difference of said first pair of floating point numbers being less than or equal to one;

and wherein a floating point-to-integer conversion operation is performed on said second floating point number in said far data path in response to said first control value indicating said floating point-to-integer conversion operation.

26. The microprocessor of claim 25, wherein said first floating point number includes a first sign bit, a first exponent value, and a first mantissa value, wherein said second floating point number includes a second sign bit, a second exponent value, and a second mantissa value, and wherein said far data path includes a exponent difference generation unit coupled to receive said first exponent value, said second exponent value, and said first control value, wherein said exponent difference generation unit is configured to generate one or more exponent difference values.

27. The microprocessor of claim 26, wherein said exponent difference generation unit is configured to generate said one or more exponent difference values usable to align said first mantissa value and said second mantissa value in response to said first control signal indicating said effective addition operation or said effective subtraction operation.

28. The microprocessor of claim 27, wherein said exponent difference generation unit is configured to generate a first integer conversion shift count within said one or more exponent difference values in response to said first control signal indicating said floating point-to-integer conversion operation, wherein said first integer conversion shift count is usable to shift said second mantissa value to a bit position within said far data path which corresponds to said second exponent value.

29. The microprocessor of claim 28, wherein said far data path includes a shift unit coupled to receive said first mantissa value, said second mantissa value, and said one or more exponent difference values, wherein said shift unit is configured to generate a shifted first mantissa value from said first mantissa value and a shifted second mantissa value from said second mantissa value according to said one or more exponent difference values.

30. The microprocessor of claim 29, wherein said shifted second mantissa value is generated from said second mantissa value according to said first integer conversion shift count in response to said first control signal indicating said floating point-to-integer conversion operation, wherein a leading one bit within said shifted second mantissa value is located in bit position having an associated exponent magnitude which corresponds to said second exponent value.

31. The microprocessor of claim 25, wherein an output of said floating point-to-integer conversion operation is clamped at a maximum representable integer in response to said second floating point number being greater than said maximum representable integer.

32. The microprocessor of claim 25, wherein an output of said floating point-to-integer conversion operation is clamped at a minimum representable integer in response to said second floating point number being less than said minimum representable integer.

33. A microprocessor, comprising:

a leading one prediction unit configured to predict a position of a leading one value within a result mantissa value corresponding to a first floating point subtraction operation performed upon a first floating point number and a second floating point number, wherein said leading one prediction unit is coupled to receive a first operand corresponding to said first floating point number and a second operand corresponding to said second floating point number, wherein said leading one prediction unit is configured to generate a prediction string including a prediction value for each bit position within said result mantissa value, wherein each prediction value within said prediction string is generated by utilizing values from a single corresponding bit position within said first operand and said second operand, wherein an indication of said position of said leading one value within said result mantissa value is given by a bit position of a most significant asserted prediction value within said prediction string;

wherein said prediction string is generated according to a prediction that said first floating point number includes a first exponent value that is one greater than a second exponent value included in said second floating point number.

34. The microprocessor of claim 33, wherein said prediction string includes a first prediction value corresponding to a most significant bit position of said prediction string, wherein said first prediction value is generated using only values from a second most significant bit position of said first operand and said second operand.

35. A method for detecting a position of a leading one value in a result mantissa value corresponding to a first floating point subtract operation performed upon a first floating point number and a second floating point number, said method comprising:

receiving a first operand corresponding to said first floating point number;

receiving a second operand corresponding to said second floating point number;

forming a prediction string by generating a prediction value for each bit position within said result mantissa value, wherein each prediction value within said prediction string is generated using values of a single corresponding bit position within said first operand and said second operand;

determining a position of said leading one value within said result mantissa value by locating a most significant asserted bit position within said prediction string;

wherein said prediction string is generated according to a prediction that a first floating point exponent value included in said first floating point number is one greater than a second floating point exponent value included in said second floating point number.

36. A method for performing effective subtraction for floating point input values having an absolute exponent difference less than or equal to one, comprising:

receiving a first mantissa portion corresponding to a first floating point input value and an inverted version of a second mantissa portion corresponding to a second floating point input value;

adding said first mantissa portion and said inverted version of said second mantissa portion in order to produce a first output value and a second output value, wherein said first output value is equal to said first mantissa portion plus said inverted version of said second mantissa portion, and wherein said second output value is equal to said first output value plus one;

generating a first plurality of preliminary selection signals indicative of either said first output value or said second output value, wherein each of said first plurality of preliminary selection signals is generated according to one of a plurality of input/output prediction values;

generating a first set of control signals which indicate which of said plurality of input/output prediction values actually occurs, wherein said first set of control signals are generated using a carry in signal corresponding to a most significant bit position of said first output value;

selecting one of said first plurality of preliminary selection signals as a final select value by utilizing said first set of control signals;

selecting either said first output value or said second output value as a preliminary subtraction result according to said final select value.

37. The method of claim 1, further comprising:

detecting that said first output value is negative;

inverting said first output value;

selecting said inverted first output value as said preliminary subtraction result.

38. A microprocessor, comprising:

a first execution unit coupled to receive a given pair of floating point input values, including:

a first close data path configured to perform effective subtraction on said given pair of floating point input values by predicting said given pair of floating point input values to have an absolute exponent difference less than or equal to one, wherein said first close data path includes:

a first close path selection unit configured to generate a first close path selection signal usable to select either a first close path adder result or a second close path adder result as a first close path preliminary subtraction result, wherein said first close path adder result is equal to a difference value of said given pair of floating point input values, and wherein said second close path adder result is equal to said first close path adder result plus one, wherein said first close path selection unit includes:

a first plurality of logic units coupled to receive a least significant bit and a guard bit corresponding to said first close path adder result, wherein each of said first plurality of logic units is configured to generate one of a first plurality of close path preliminary select signals, wherein each of said first plurality of close path preliminary select signals corresponds to a different set of predictions regarding said given pair of floating point input values and said first close path adder result;

a first close path selection multiplexer coupled to receive said first plurality of close path preliminary select signals, wherein said first close path selection multiplexer is configured to select said first close path selection signal in response to receiving a first plurality of control signals, wherein said first plurality of control signals include a first control signal and a second control signal generated in said first close path selection unit in response to receiving a carry in signal for a most significant bit position of said first close path adder result, wherein said first control signal is indicative of a sign value of said first close path adder result, and wherein said second control signal is indicative of a most significant bit of said first close path adder result.

39. The microprocessor of claim 38, wherein an output of said first close data path is discarded if said absolute exponent difference of said given pair of floating point input values is calculated to be greater than one.

40. The microprocessor of claim 38, wherein selection of said first close path adder result or said second close path adder result effectuates a round-to-nearest-number operation for a result of said effective subtraction of said given pair of floating point input values.

41. The microprocessor of claim 38, wherein said first close data path is configured to generate a first close path result in response to receiving said given pair of floating point input values.

42. A microprocessor, comprising:

an execution unit coupled to receive a first pair of floating point input values, including:

a close data path configured to perform a first effective subtract operation on said first pair of floating point input values if said first pair of floating point input values have an absolute exponent difference less than or equal to one, wherein said close data path includes:

a first arithmetic unit configured to generate a first difference value and a second difference value, wherein said first difference value is equal to a difference of mantissa portions of said first pair of floating point input values, and wherein said second difference value is equal to said first difference value plus one;

a first multiplexer unit coupled to receive said first output value and said second output value, wherein said first multiplexer unit is configured to select either said first output value or said second output value as a preliminary subtraction result according to a close path selection signal;

a first selection unit configured to generate said close path selection signal from a plurality of preliminary selection signals, wherein said first selection unit utilizes a carry in signal to a most significant bit position of said first arithmetic unit in order to select one of said plurality of preliminary selection signals as said close path selection signal.

43. The microprocessor of claim 42, wherein selection of either said first output value or said second output value is usable to effectuate a round-to-nearest operation on a result of said first effective subtract operation.

44. The microprocessor of claim 42, wherein, if said first difference value is calculated to be negative, said multiplexer unit is configured to convey an inverted version of said first difference value as said preliminary subtraction result.

45. The microprocessor of claim 42, wherein said first selection unit utilizes a least significant bit and a guard bit corresponding to said first output value in order to generate said plurality of preliminary selection signals.

46. The microprocessor of claim 45, wherein said plurality of preliminary selection signals includes a first select signal corresponding to a prediction that exponent values of said first pair of floating point input values are equal and said first output value is negative.

47. A look-up table for determining output values for a first mathematical function and a second mathematical function, said look-up table comprising:

a first plurality of storage locations configured to store a first plurality of base values for said first mathematical function and a second plurality of base values for said second mathematical function;

a second plurality of storage locations configured to store a first plurality of difference values for said first mathematical function and a second plurality of difference values for said second mathematical function;

an address control unit coupled to receive a first set of input signals indicative of a first input value to said look-up table and whether a first output value corresponding to said first input value is to be generated for said first mathematical function or said second mathematical function, wherein said address control unit is configured to generate a first address value from said first set of input signals and convey said first address value to said first plurality of storage locations and said second plurality of storage locations, and wherein said first plurality of storage locations is configured to output a first base value in response to receiving said first address value, and wherein said second plurality of storage locations is configured to output a first difference value in response to receiving said first address value;

an output unit coupled to receive said first base value from said first plurality of storage locations and said first difference value from said second plurality of storage locations, wherein said output unit is configured to generate said first output value from said first base value and said first difference value.

48. The look-up table of claim 47, wherein said first base value is selected from said first plurality of base values and said first difference value is selected from said first plurality of difference values in response to said first set of control values including an indication that said first output value is to be generated for said first mathematical function.

49. The look-up table of claim 47, wherein said first base value is selected from said second plurality of base values and said first difference value is selected from said second plurality of difference values in response to said first set of control values including an indication that said first output value is to be generated for said second mathematical function.

50. The look-up table of claim 48, wherein each of said first plurality of base values is an output value of said first mathematical function for a corresponding one of a plurality of first function input regions, wherein each of said plurality of first function input regions is located within a predetermined first input range.

51. The look-up table of claim 50, wherein each of said first plurality of difference values is an output value difference from one of said first plurality of base values, wherein each of said first plurality of difference values is usable with said one of said first plurality of base values to determine output values of said first mathematical function for input values within selected regions of said first input range.

52. The look-up table of claim 51, wherein each of said second plurality of base values is an output value of said second mathematical function for a corresponding one of a plurality of second function input regions, wherein each of said plurality of second function input regions is located within a predetermined second input range.

53. The look-up table of claim 49, wherein said look-up table is usable to compute values of said first mathematical function for a predetermined first range of input values, and wherein said look-up table is usable to compute values of said second mathematical function for a predetermined second range of input values.

54. The look-up table of claim 53, wherein said first range of input values and said second range of input values are each divided into intervals of continuous input ranges, wherein said first range of input values is divided into a first plurality of intervals, and wherein said second range of input values is divided into a second plurality of intervals.

55. The look-up table of claim 54, wherein said first plurality of intervals and said second plurality of intervals are each divided into subintervals of continuous input ranges, wherein said first plurality of intervals includes a first plurality of subintervals, and wherein said second plurality of intervals includes a second plurality of subintervals.

56. The look-up table of claim 55, wherein said first plurality of subintervals and said second plurality of subintervals are each divided into sub-subintervals of continuous input ranges, wherein said first plurality of subintervals includes a first plurality of sub-subintervals, and wherein said second plurality of subintervals includes a second plurality of sub-subintervals.

57. The look-up table of claim 56, wherein each of said first plurality of difference values is an output value difference of said first mathematical function which corresponds to a first group of sub-subintervals within a first interval which includes said corresponding one of said first plurality of subintervals.

58. The look-up table of claim 49, wherein said output unit is configured to add said first base value and said first difference value in order to generate said first output value, and wherein said output unit is coupled

to receive a rounding constant, and wherein said output unit is configured to add said rounding constant to said first base value and said first difference value in order to generate said first output value.

59. The look-up table of claim 49, wherein said first mathematical function is f(x)=1/x, and said second mathematical function is f(x)=1/sqrt(x).

60. A look-up table for determining output values for a plurality of mathematical functions, said look-up table comprising:

a first plurality of storage locations configured to store a plurality of base values for each of said plurality of mathematical functions;

a second plurality of storage locations configured to store a plurality of difference values for each of said mathematical functions;

an address control unit coupled to receive a first set of input signals indicative of a first input value to said look-up table and a selected one of said plurality of mathematical functions, wherein a first output value is to be generated for said selected one of said plurality of mathematical functions from said first input value, wherein said address control unit is configured to generate a first address value from said first set of input signals and convey said first address value to said first plurality of storage locations and said second plurality of storage locations, and wherein said first plurality of storage locations is configured to output a first base value in response to receiving said first address value, and wherein said second plurality of storage locations is configured to output a first difference value in response to receiving said first address value;

an output unit coupled to receive said first base value from said first plurality of storage locations and said first difference value from said second plurality of storage locations, wherein said output unit is configured to generate said first output value from said first base value and said first difference value.

61. The look-up table of claim 60, wherein first base value is selected from a plurality of base values which correspond to said selected one of said plurality of mathematical functions, and wherein said first difference value is selected from a plurality of difference values which correspond to said selected one of said plurality of mathematical functions.

TITLE: **Multifunction Floating Point Addition/Subtraction Pipeline And Bipartite Look-up Table**

ABSTRACT OF THE DISCLOSURE

An optimized multimedia execution unit configured to perform vectored floating point and integer instructions. In one embodiment, the execution unit includes an add/subtract pipeline having far and close data paths. The far data path is configured to handle effective addition operations, as well as effective subtraction operations for operands having an absolute exponent difference greater than one. The close data path, conversely, is configured to handle effective subtraction operations for operands having an absolute exponent difference less than or equal to one. The close data path includes an adder unit configured to generate a first and second output value. The first output value is equal to the first input operand plus an inverted version of the second input operand, while the second output value is equal to the first output value plus one. The two output values are conveyed to a multiplexer unit, which selects one of the output values as a preliminary subtraction result based on a final selection signal received from a selection unit. The selection unit generates the final selection signal from a plurality of preliminary selection signals based on the carry in signal to the most significant bit of the first adder output value. Selection of the first or second output value in the close data path effectuates the round-to-nearest operation for the output of the adder. The execution unit may also be configured, in another embodiment, to perform floating point-to-integer and integer-to-floating point conversions. The floating point-to-integer conversions may be efficiently executed in the far data path of the add/subtract pipeline, with the integer-to-floating point instructions executed in the close data path. The execution unit may also include a plurality of add/subtract pipelines, allowing vectored add, subtract, and integer/floating point conversion instructions to be performed. The execution unit be also expanded to handle additional arithmetic instructions (such as reverse subtract and accumulate functions) by appropriate input multiplexing. Finally, functions like extreme value (minimum/maximum) and comparison instructions may also be implemented by proper multiplexing of output results. A method for generating entries for a bipartite look-up table having base and difference table portions is also disclosed. In one embodiment, these entries are usable to form output values for a mathematical function, f(x), in response to receiving corresponding input values within a predetermined input range. The method first comprises partitioning the input range into I intervals, J subintervals/interval, and K sub-subintervals/subinterval. For a given interval M, the method includes generating K difference table entries and J base table entries. Each of the K difference table entries corresponds to a particular group of sub-subintervals within interval M, each of which has the same relative position within their respective subintervals. Each difference table entry is computed by averaging difference values for the sub-subintervals included in a corresponding group N. Each difference value which makes up this average is equal to f(X1)-f(X2), where X1 is the midpoint of the sub-subinterval within group N, and X2 is the midpoint of a predetermined reference sub-subinterval within the same subinterval as X1. Each of these midpoints is calculated such that maximum absolute error is minimized for all possible input values in the sub-subinterval. Each of the J base table entries, on the other hand, corresponds to a subinterval within interval M. Each entry is equal to f(X2)+adjust, where X2 is the midpoint of the reference sub-subinterval of the subinterval corresponding to the base table entry. The adjust value is calculated so that error introduced by the averaging of the difference table entries is evenly distributed over the entire subinterval. A multi-function

look-up table for determining output values for predetermined ranges of a first mathematical function and a second mathematical function. In one embodiment, the multi-function look-up table is a bipartite look-up table including a first plurality of storage locations and a second plurality of storage locations. The first plurality of storage locations store base values for the first and second mathematical functions. Each base value is an output value (for either the first or second function) corresponding to an input region which includes the look-up table input value. The second plurality of storage locations, on the other hand, store difference values for both the first and second mathematical functions. These difference values are used for linear interpolation in conjunction with a corresponding base value in order to generate a look-up table output value. The multi-function look-up table further includes an address control unit coupled to receive a first input value and a signal which indicates whether an output value is to be generated for the first or second mathematical function. The address control unit then generates a first address value from these signals which is in turn conveyed to the first and second plurality of storage locations. In response to receiving the first address value, the first and second plurality of storage locations are configured to output a first base value and a first difference value, respectively. The first base value and first difference value are then conveyed to an output unit configured to generate a look-up table output value from the two values.

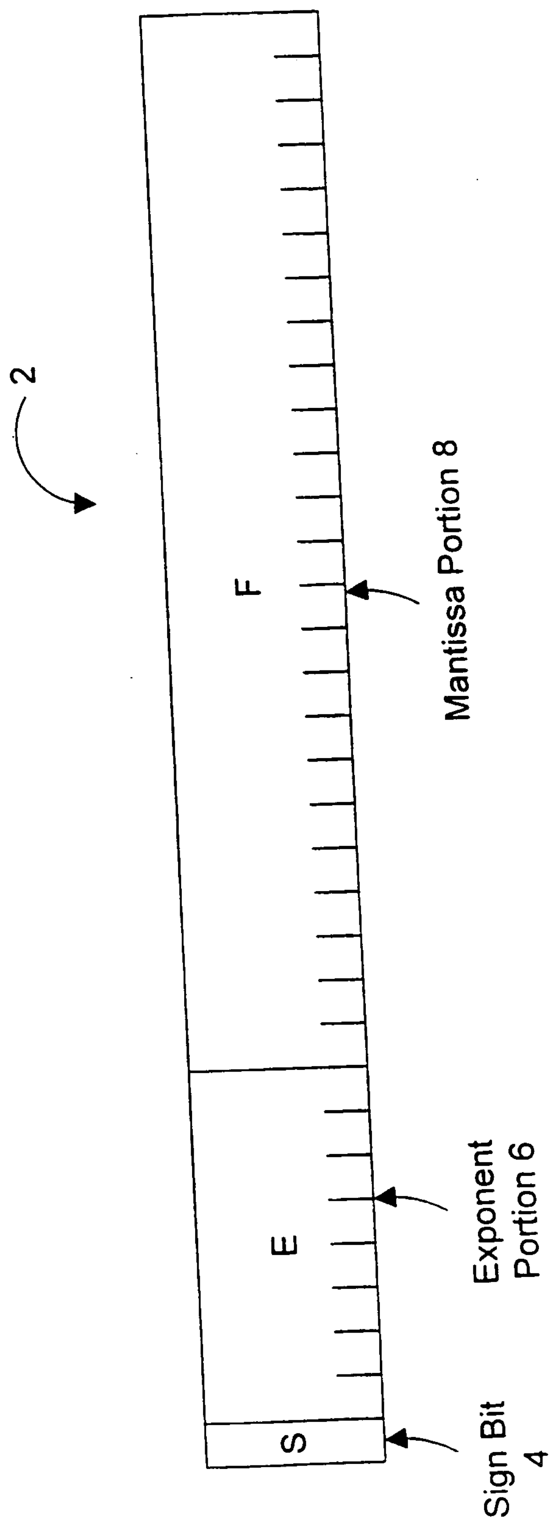


FIG. 1
(PRIOR ART)

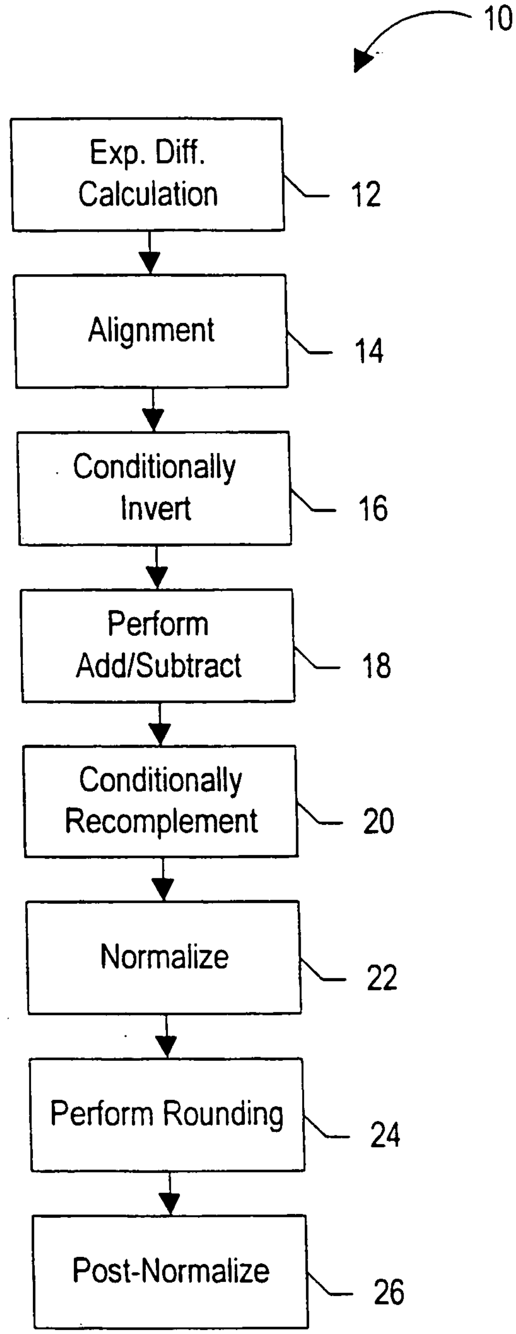


FIG. 2
(PRIOR ART)

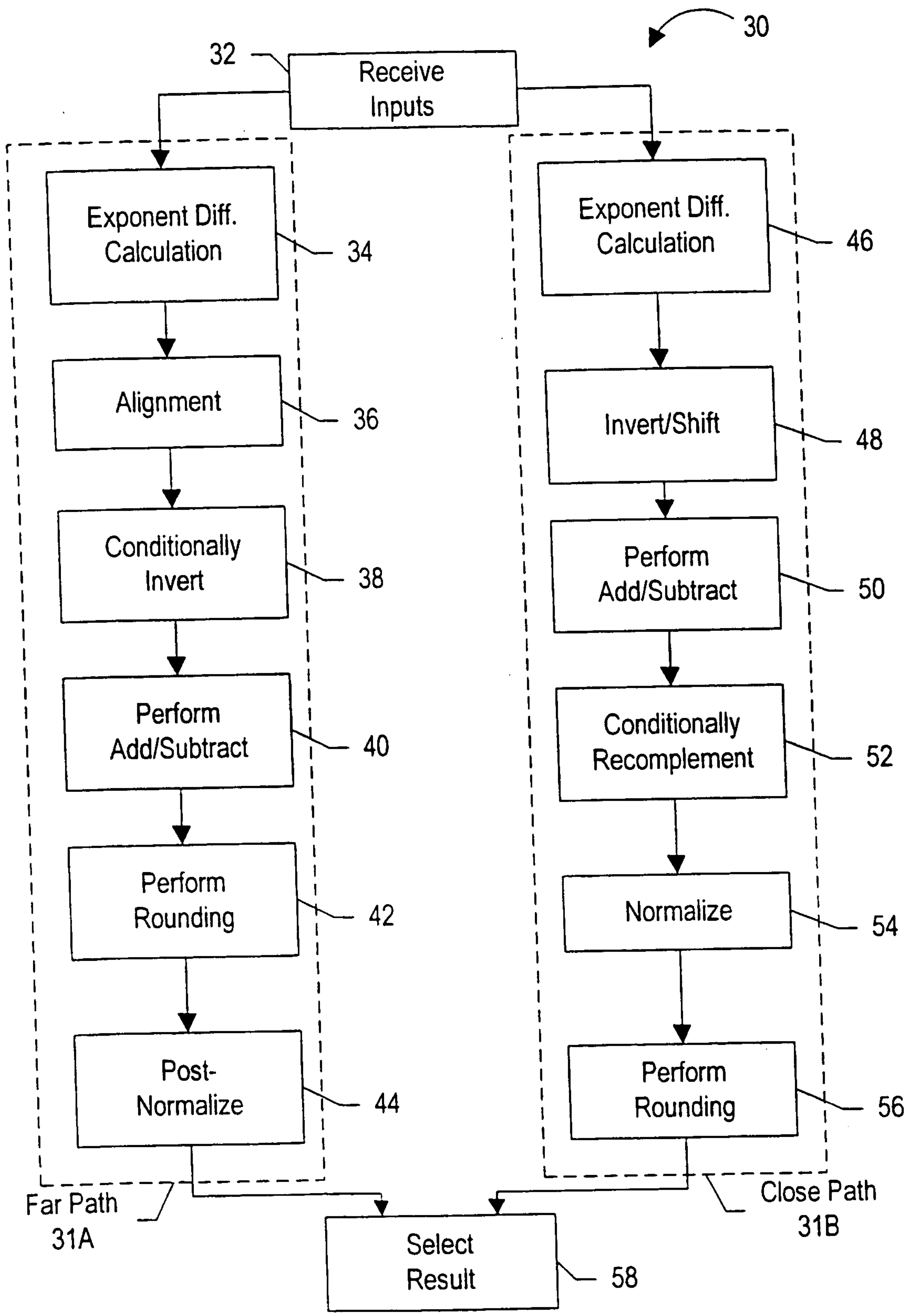


FIG. 3
(PRIOR ART)

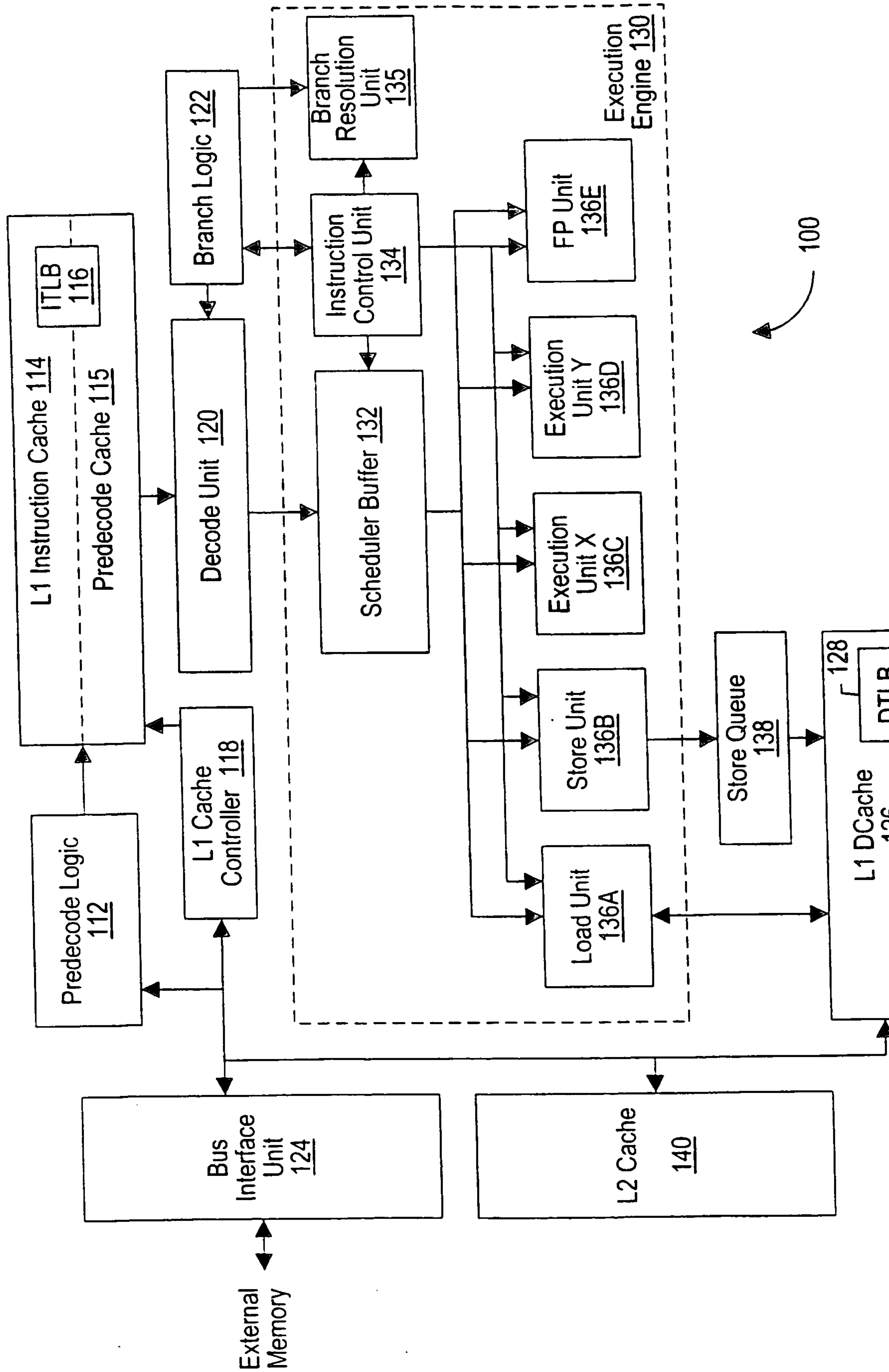

Branch Resolution Unit 135
Execution Engine 130
Branch Logic 122
Instruction Control Unit 134
FP Unit 136E
L1 Instruction Cache 114
ITLB 116
Predecode Cache 115
Decode Unit 120
Scheduler Buffer 132
Execution Unit Y 136D
Execution Unit X 136C
Store Unit 136B
Store Queue 138
128
DTLB
L1 DCache 126
Load Unit 136A
Predecode Logic 112
L1 Cache Controller 118
Bus Interface Unit 124
L2 Cache 140
External Memory
100

FIG. 4

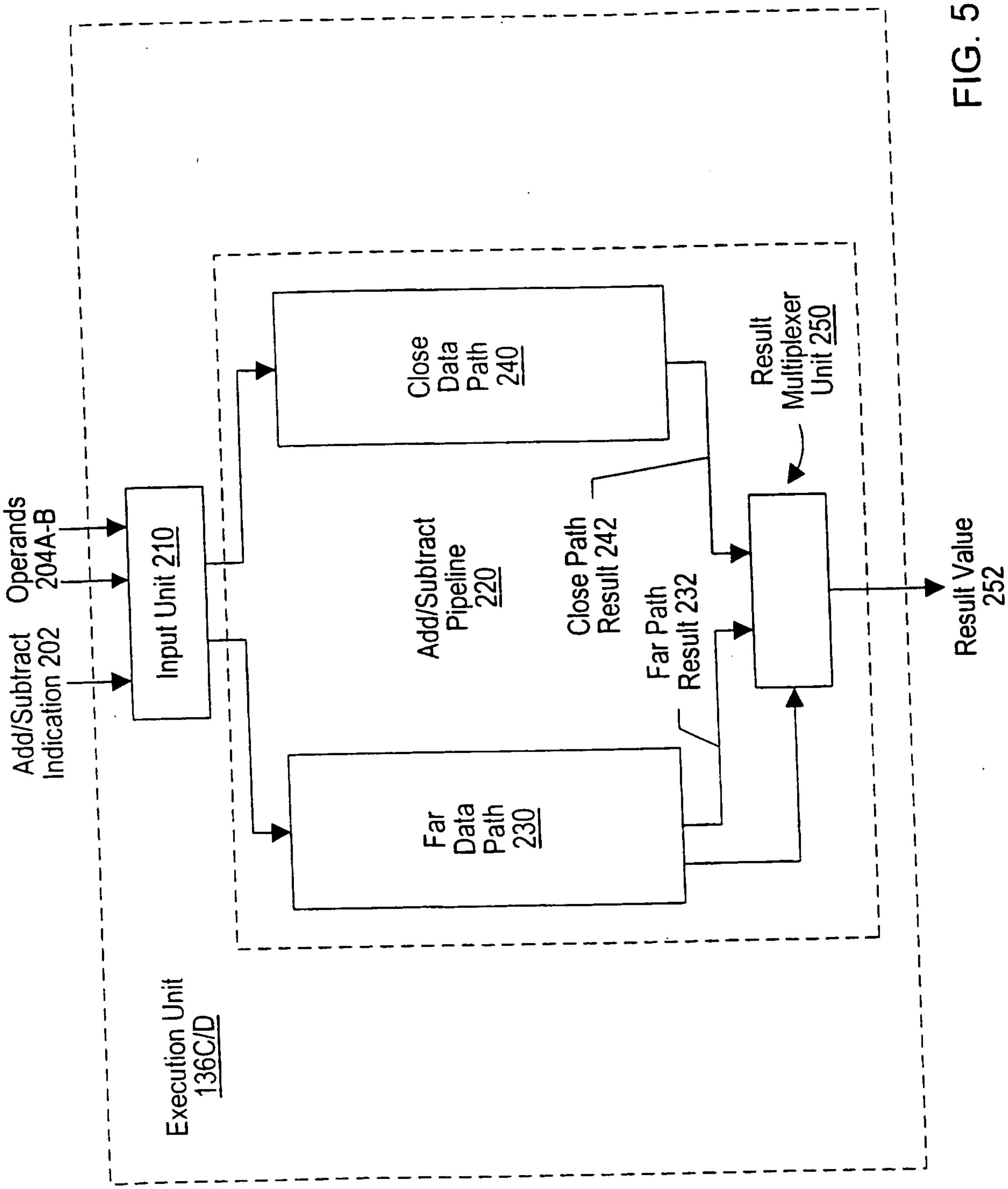

Execution Unit 136C/D
Add/Subtract Indication 202
Operands 204A-B
Input Unit 210
Add/Subtract Pipeline 220
Close Data Path 240
Far Data Path 230
Close Path Result 242
Far Path Result 232
Result Multiplexer Unit 250
Result Value 252

FIG. 5

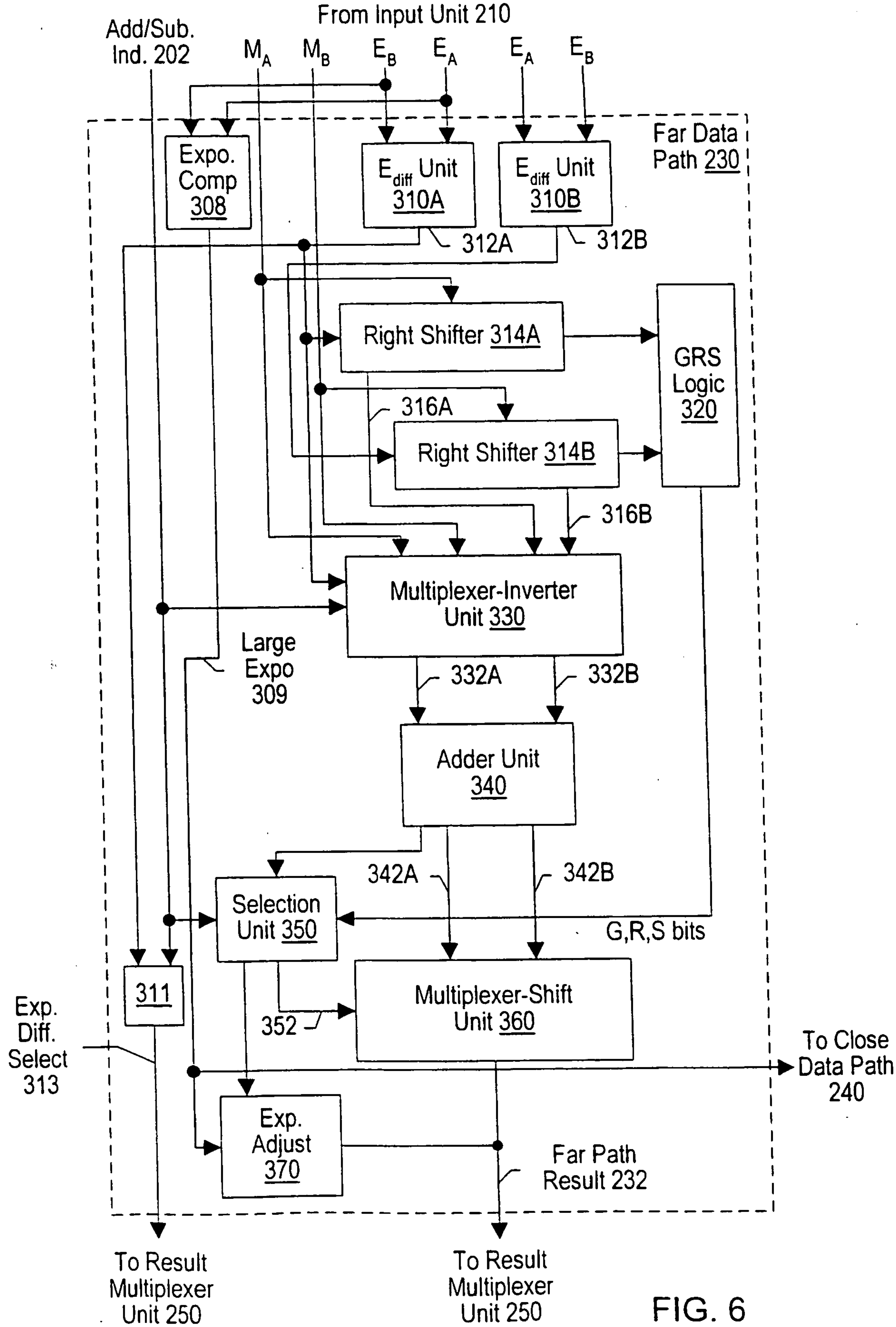

From Input Unit 210
Add/Sub. Ind. 202
M_A
M_B
E_B
E_A
E_A
E_B
Far Data Path 230
Expo. Comp 308
E_{diff} Unit 310A
E_{diff} Unit 310B
312A
312B
Right Shifter 314A
GRS Logic 320
316A
Right Shifter 314B
316B
Multiplexer-Inverter Unit 330
Large Expo 309
332A
332B
Adder Unit 340
342A
342B
Selection Unit 350
G,R,S bits
311
Exp. Diff. Select 313
352
Multiplexer-Shift Unit 360
To Close Data Path 240
Exp. Adjust 370
Far Path Result 232
To Result Multiplexer Unit 250
To Result Multiplexer Unit 250

FIG. 6

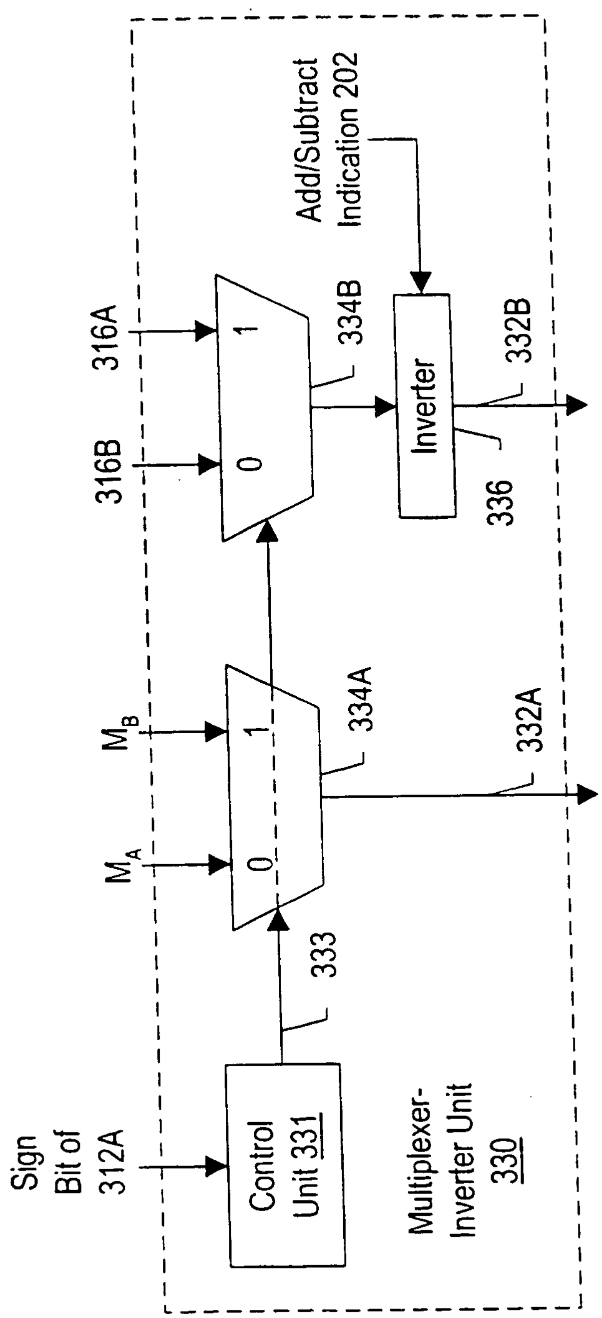

Add/Subtract
Indication 202
316A
316B
1
0
334B
Inverter
332B
336
M_B
M_A
1
0
334A
332A
333
Sign
Bit of
312A
Control
Unit 331
Multiplexer-
Inverter Unit
330

FIG. 7

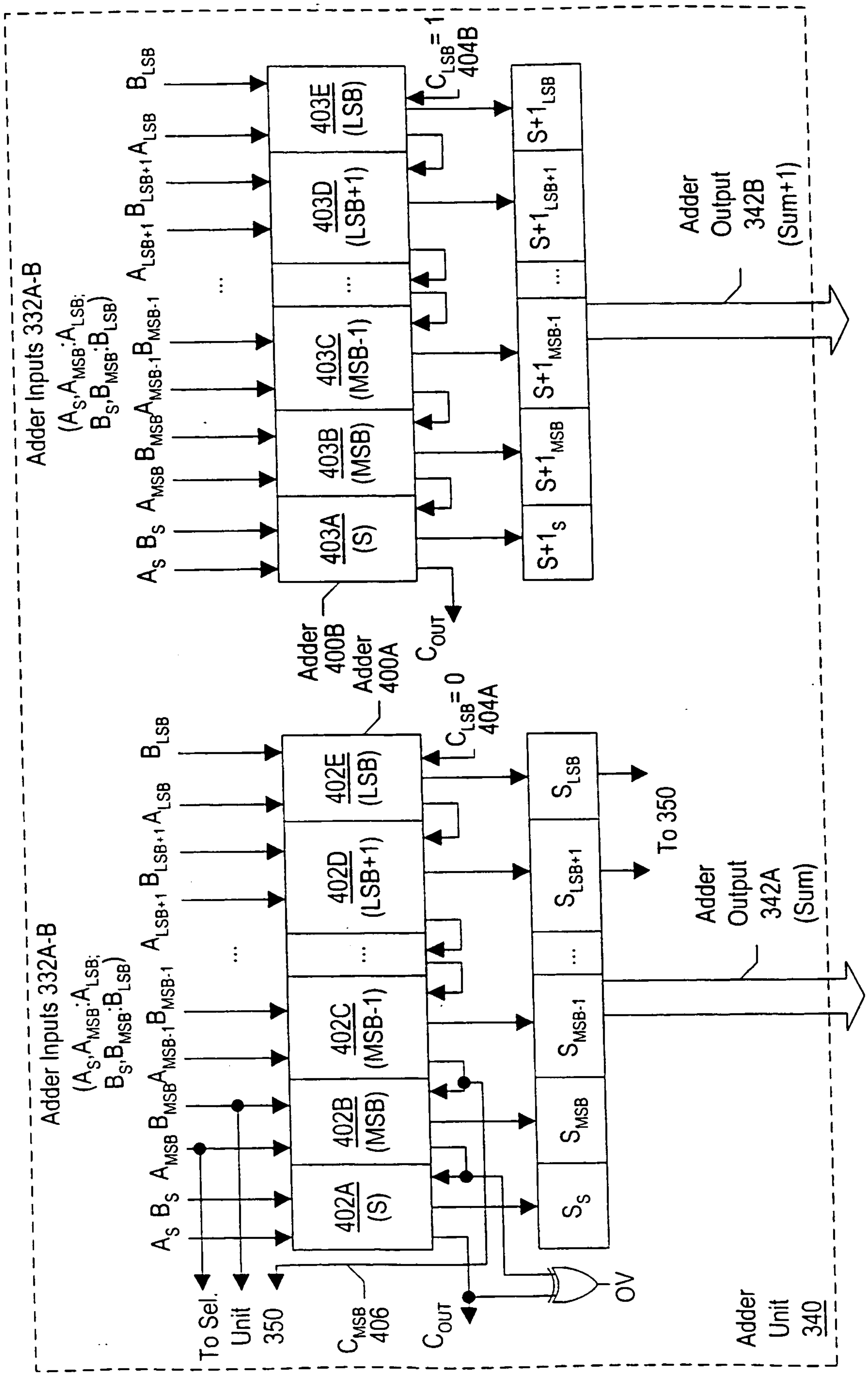
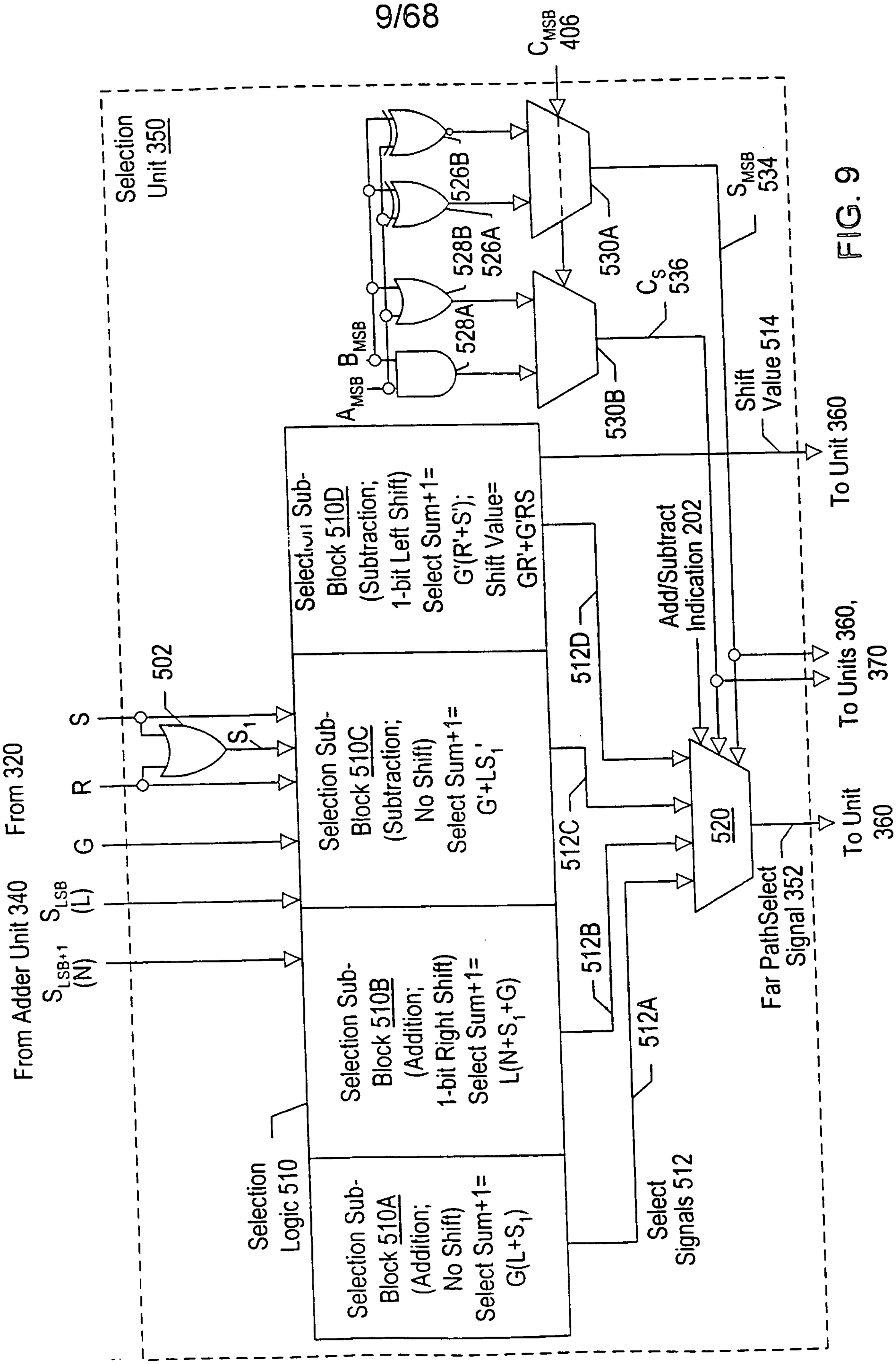

Adder Inputs 332A-B
(A_S,A_{MSB}:A_{LSB};
B_S,B_{MSB}:B_{LSB})
403A (S)
403B (MSB)
403C (MSB-1)
403D (LSB+1)
403E (LSB)
C_{LSB}= 1
404B
$S+1_S$
$S+1_{MSB}$
$S+1_{MSB-1}$
$S+1_{LSB+1}$
$S+1_{LSB}$
Adder Output 342B (Sum+1)
Adder 400B
Adder 400A
C_{OUT}
402A (S)
402B (MSB)
402C (MSB-1)
402D (LSB+1)
402E (LSB)
C_{LSB}= 0
404A
S_S
S_{MSB}
S_{MSB-1}
S_{LSB+1}
S_{LSB}
To 350
Adder Output 342A (Sum)
To Sel. Unit 350
C_{MSB} 406
OV
Adder Unit 340

FIG. 8



Selection Unit 350
C_{MSB} 406
A_{MSB} B_{MSB}
526B
528B
526A
528A
530A
530B
S_{MSB} 534
C_S 536
Shift Value 514
To Unit 360
Selection Sub-Block 510D (Subtraction; 1-bit Left Shift) Select Sum+1= G'(R'+S'); Shift Value= GR'+G'RS
Selection Sub-Block 510C (Subtraction; No Shift) Select Sum+1= G'+LS$_1$'
Selection Sub-Block 510B (Addition; 1-bit Right Shift) Select Sum+1= L(N+S$_1$+G)
Selection Sub-Block 510A (Addition; No Shift) Select Sum+1= G(L+S$_1$)
Selection Logic 510
From Adder Unit 340
S_{LSB+1} (N)
S_{LSB} (L)
From 320
G
R
S
502
S_1
512D
512C
512B
512A
Select Signals 512
Add/Subtract Indication 202
To Units 360, 370
520
To Unit 360
Far PathSelect Signal 352

FIG. 9

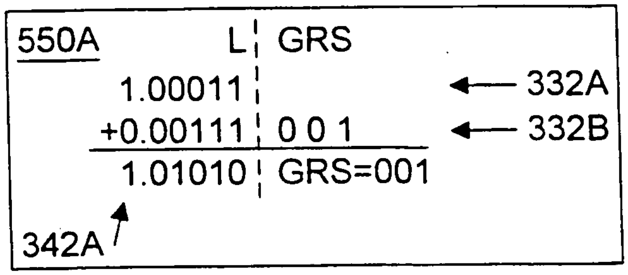


FIG. 10A

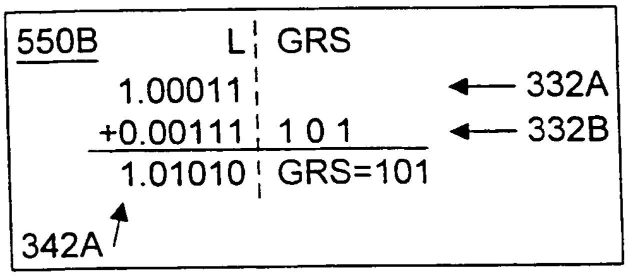


FIG. 10B

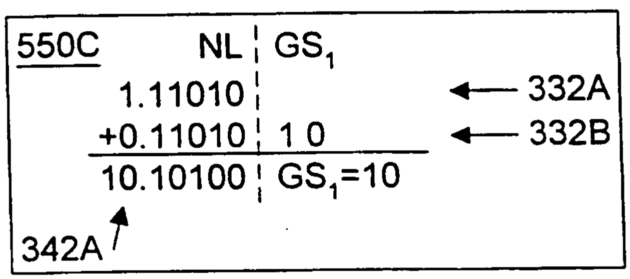


FIG. 10C

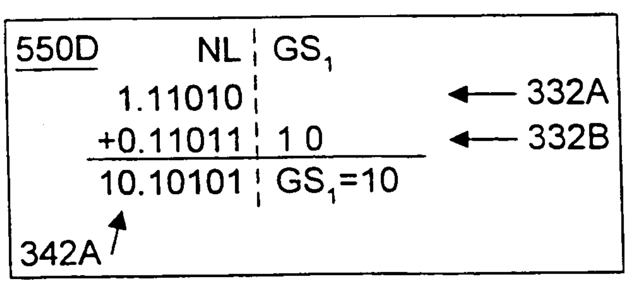


FIG. 10D

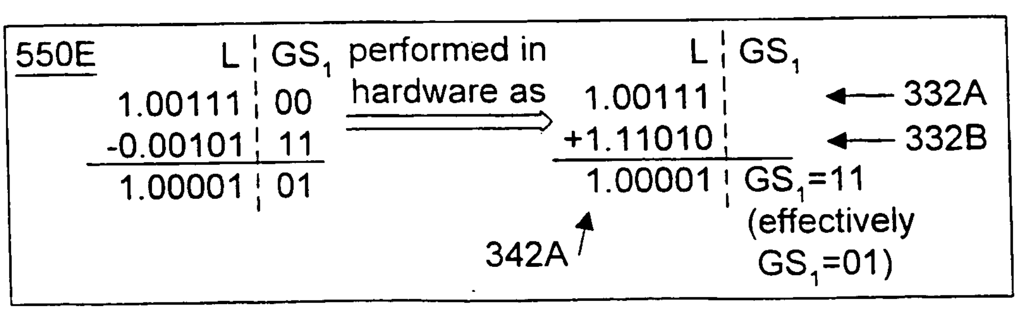


FIG. 10E

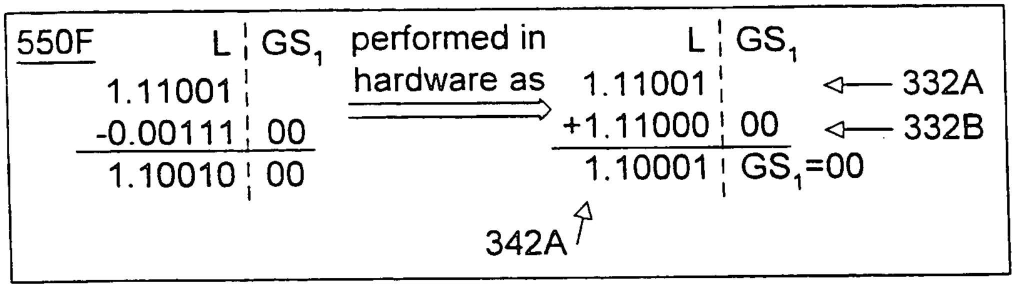


FIG. 10F

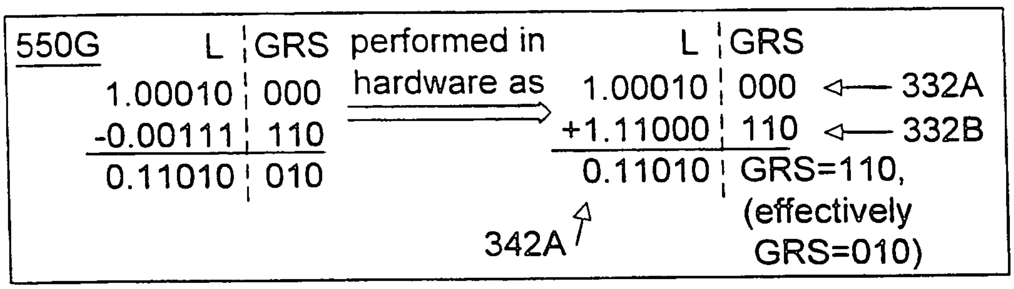


FIG. 10G

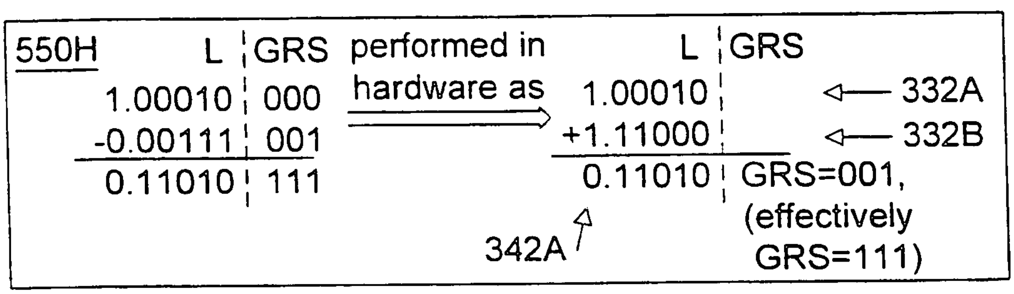


FIG. 10H

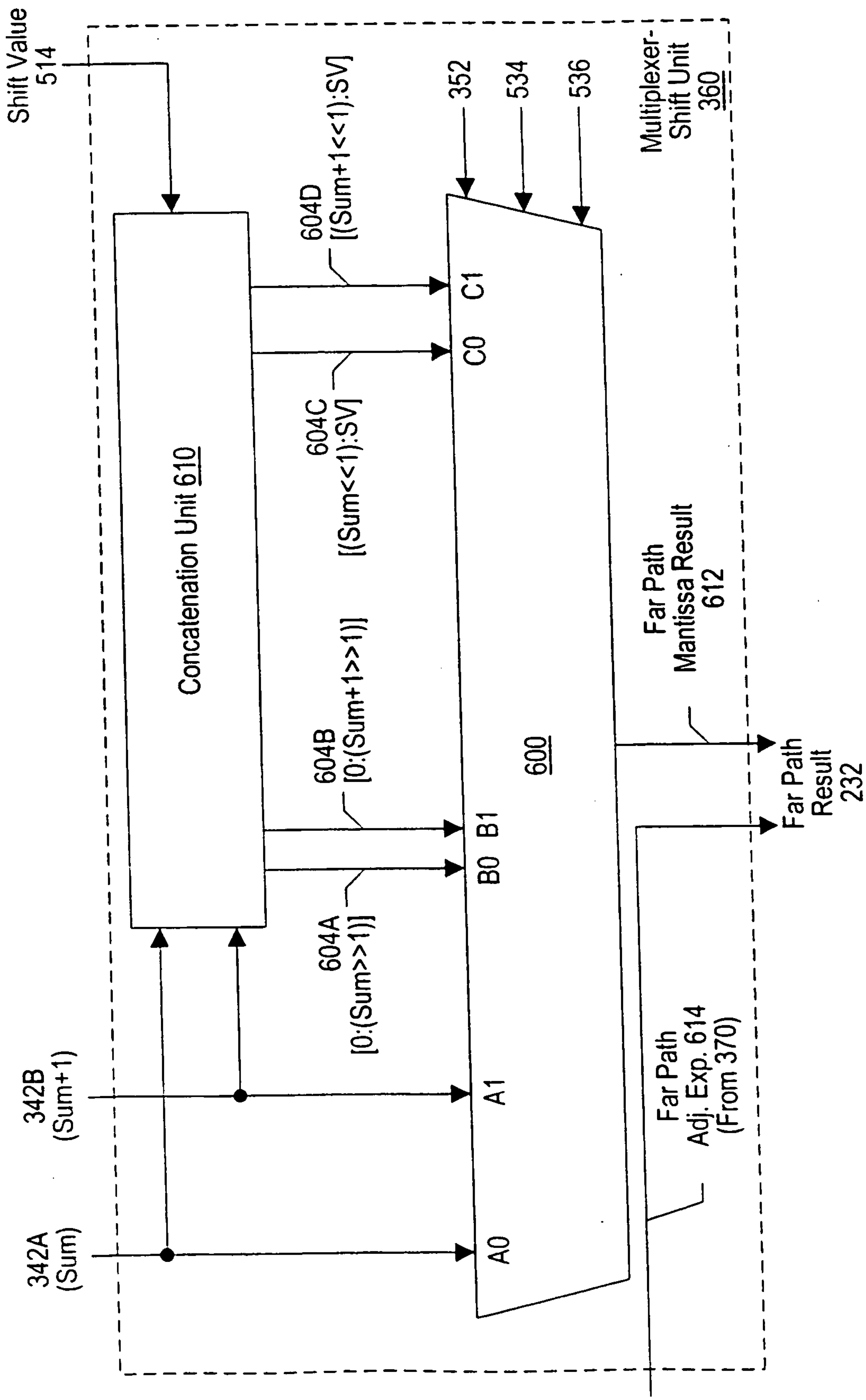
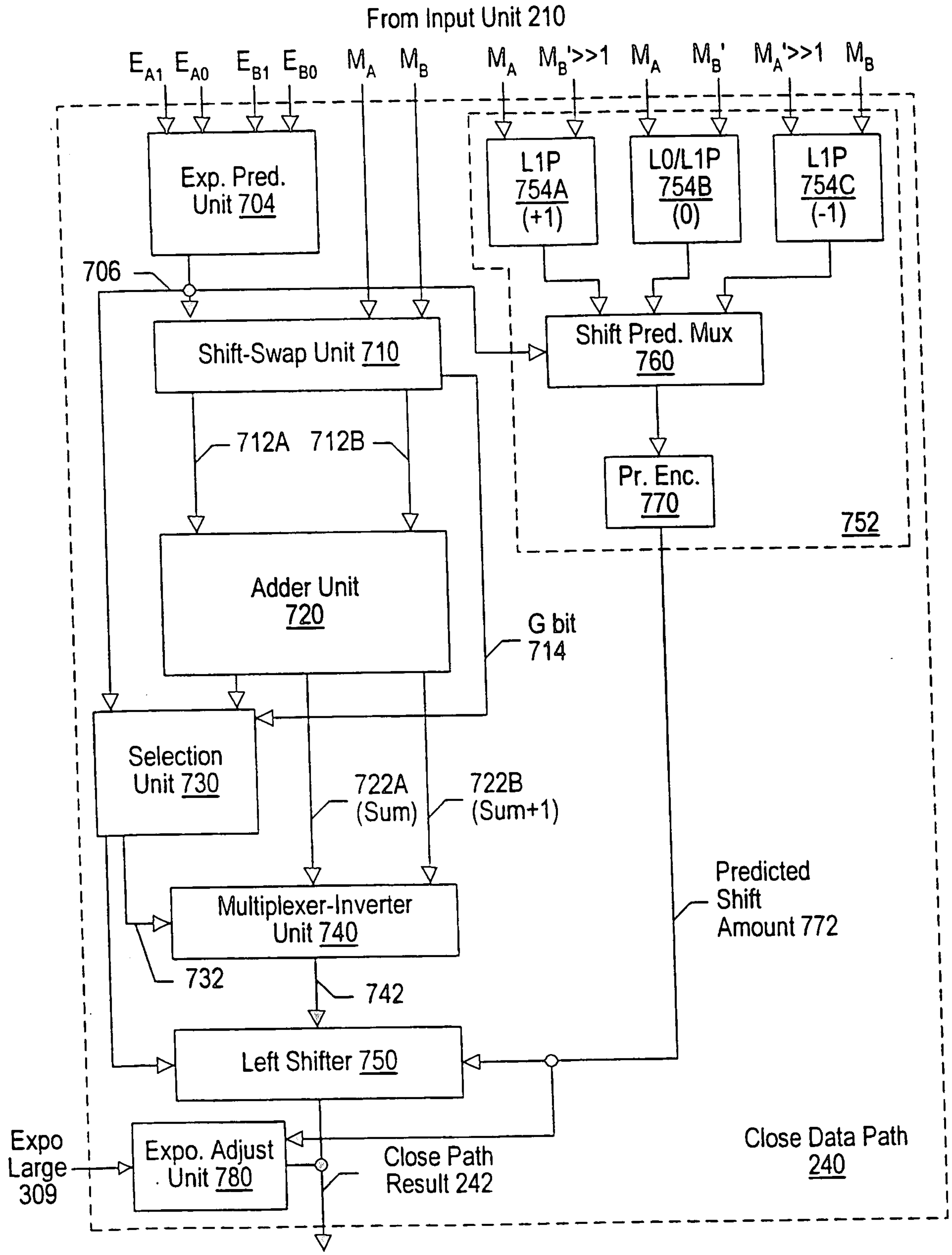

Shift Value
514
Multiplexer-
Shift Unit
360
352
534
536
604D
[(Sum+1<<1):SV]
C1
C0
604C
[(Sum<<1):SV]
Concatenation Unit 610
604B
[0:(Sum+1>>1)]
B1
B0
604A
[0:(Sum>>1)]
600
Far Path
Mantissa Result
612
Far Path
Result
232
A1
A0
342B
(Sum+1)
342A
(Sum)
Far Path
Adj. Exp. 614
(From 370)

FIG. 11



From Input Unit 210
E_{A1} E_{A0} E_{B1} E_{B0} M_A M_B M_A M_B'>>1 M_A M_B' M_A'>>1 M_B
Exp. Pred. Unit 704
L1P 754A (+1)
L0/L1P 754B (0)
L1P 754C (-1)
706
Shift-Swap Unit 710
Shift Pred. Mux 760
712A 712B
Pr. Enc. 770
752
Adder Unit 720
G bit 714
Selection Unit 730
722A (Sum)
722B (Sum+1)
Predicted Shift Amount 772
Multiplexer-Inverter Unit 740
732
742
Left Shifter 750
Expo Large 309
Expo. Adjust Unit 780
Close Path Result 242
Close Data Path 240

FIG. 12

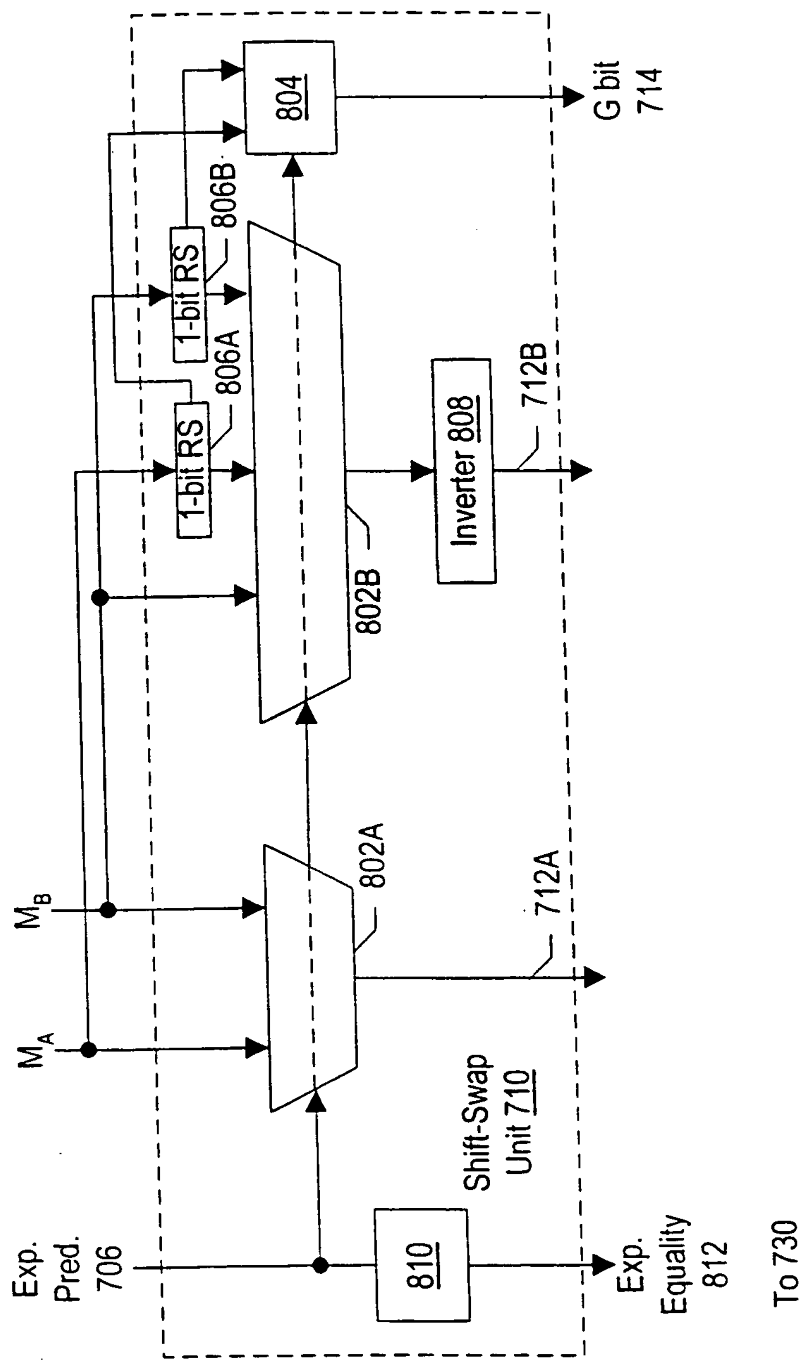

804
G bit
714
806B
1-bit RS
806A
1-bit RS
712B
Inverter 808
802B
802A
712A
M_B
M_A
Shift-Swap
Unit 710
Exp.
Pred.
706
810
Exp.
Equality
812
To 730

FIG. 13

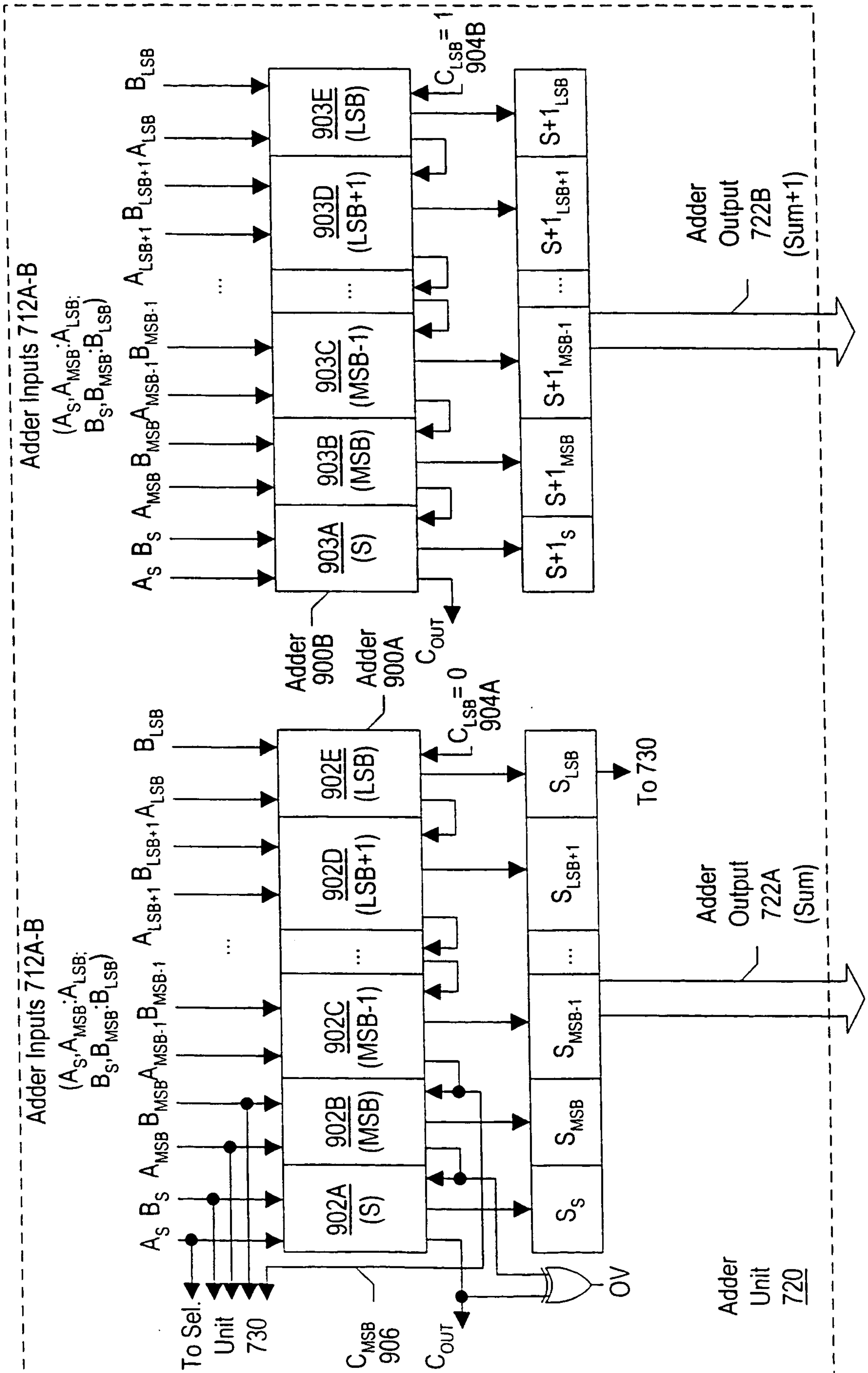
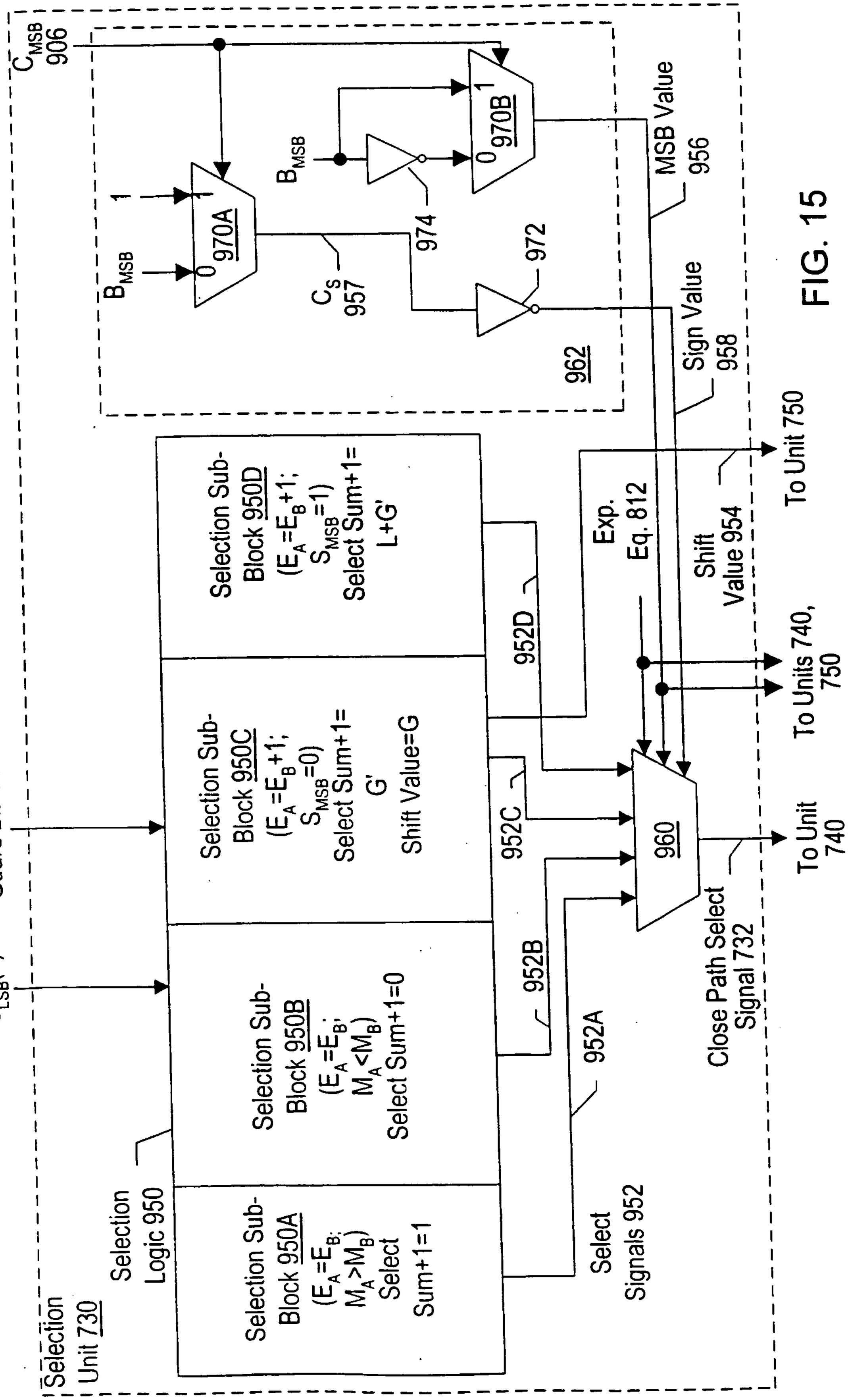

Adder Inputs 712A-B
(A_S,A_{MSB}:A_{LSB};
B_S,B_{MSB}:B_{LSB})
A_S B_S A_{MSB} B_{MSB} A_{MSB-1} B_{MSB-1} ... A_{LSB+1} B_{LSB+1} A_{LSB} B_{LSB}
903A (S)
903B (MSB)
903C (MSB-1)
903D (LSB+1)
903E (LSB)
C_{LSB} = 1 904B
$S+1_S$ $S+1_{MSB}$ $S+1_{MSB-1}$ ... $S+1_{LSB+1}$ $S+1_{LSB}$
Adder Output 722B (Sum+1)
Adder 900B
Adder 900A
C_{OUT}
902A (S)
902B (MSB)
902C (MSB-1)
902D (LSB+1)
902E (LSB)
C_{LSB} = 0 904A
S_S S_{MSB} S_{MSB-1} ... S_{LSB+1} S_{LSB}
To 730
Adder Output 722A (Sum)
OV
To Sel. Unit 730
C_{MSB} 906
C_{OUT}
Adder Unit 720

FIG. 14



C_{MSB} 906
B_{MSB}
1
0
970A
970B
974
972
C_S 957
962
MSB Value 956
Sign Value 958
Selection Sub-Block 950D ($E_A=E_B+1$; $S_{MSB}=1$) Select Sum+1= L+G'
Selection Sub-Block 950C ($E_A=E_B+1$; $S_{MSB}=0$) Select Sum+1= G' Shift Value=G
Selection Sub-Block 950B ($E_A=E_B$; $M_A<M_B$) Select Sum+1=0
Selection Sub-Block 950A ($E_A=E_B$; $M_A>M_B$) Select Sum+1=1
952D
952C
952B
952A
960
Exp. Eq. 812
Shift Value 954
To Unit 750
To Units 740, 750
To Unit 740
Close Path Select Signal 732
Select Signals 952
Selection Logic 950
Selection Unit 730
From Shift-Swap Unit 710 Guard Bit 714
From Adder Unit 720 S_{LSB}(L)

FIG. 15

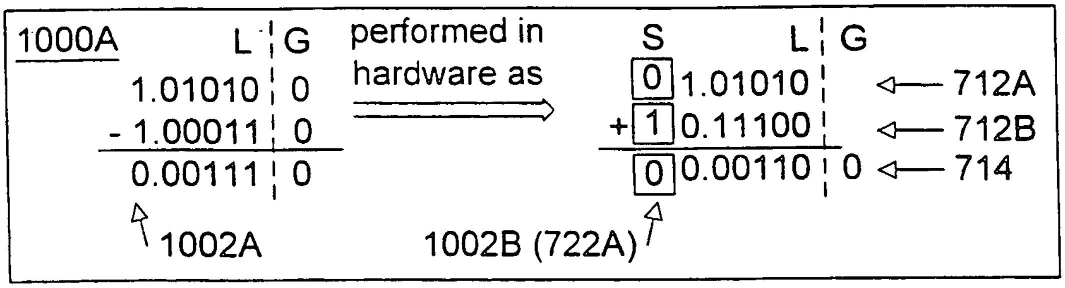


FIG. 16A

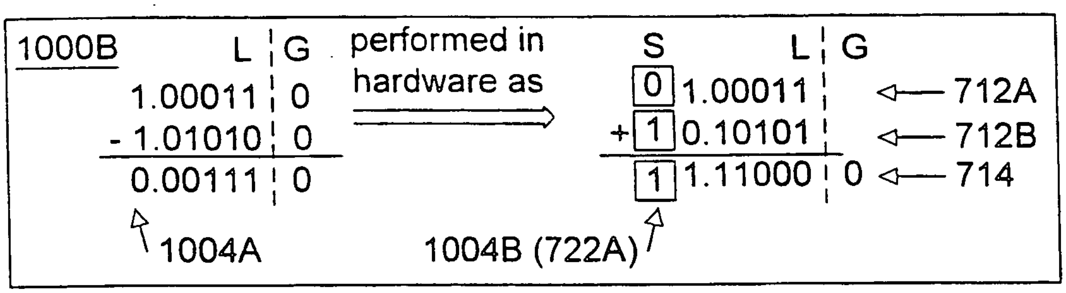


FIG. 16B

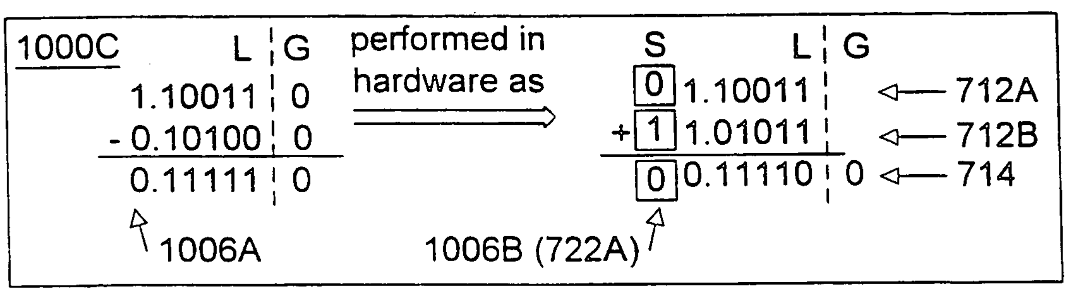


FIG. 16C

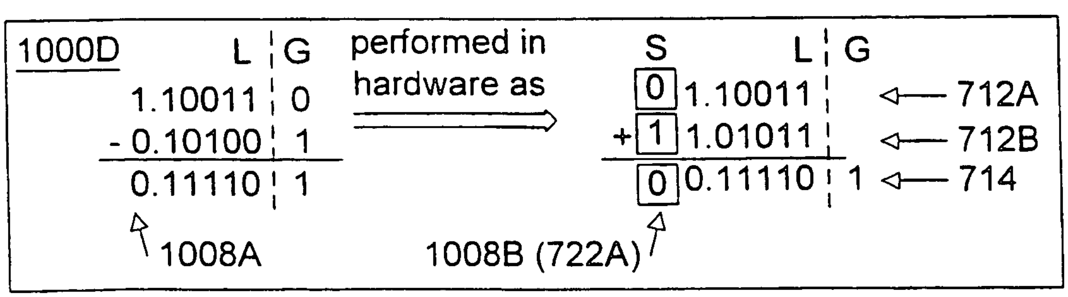


FIG. 16D

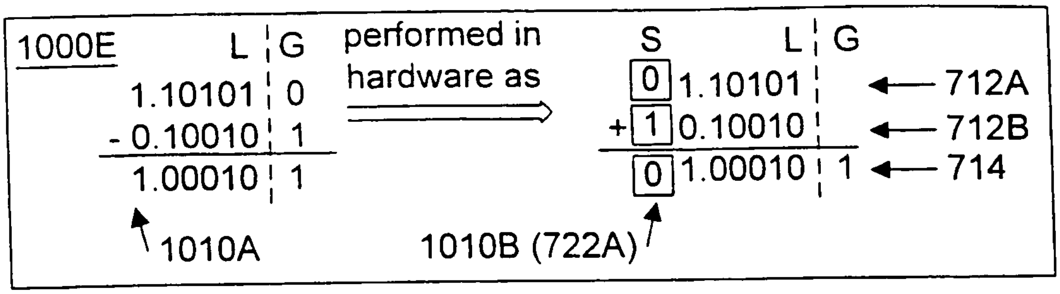


FIG. 16E

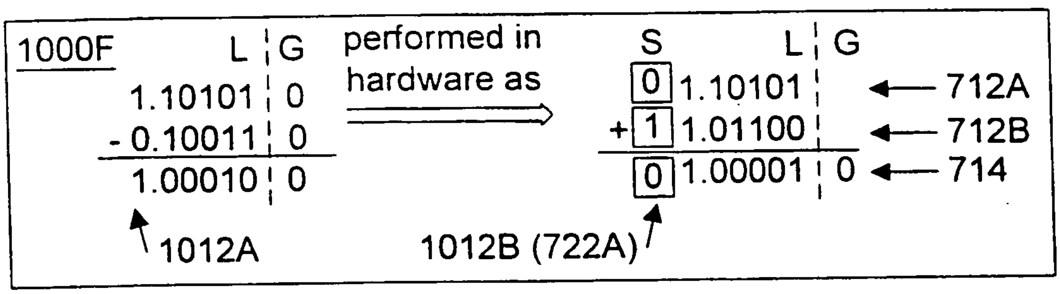


FIG. 16F

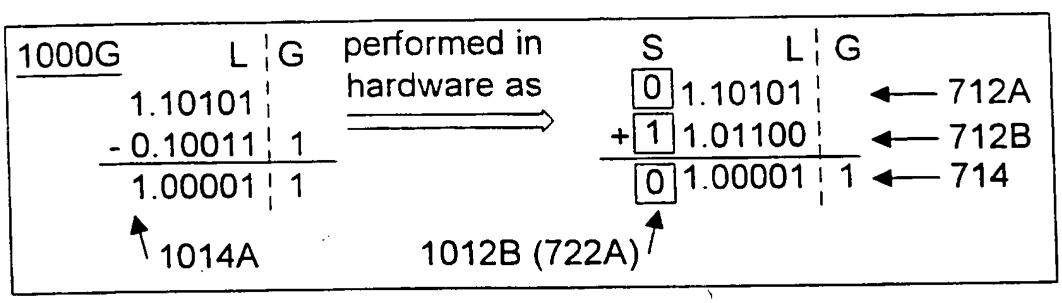


FIG. 16G

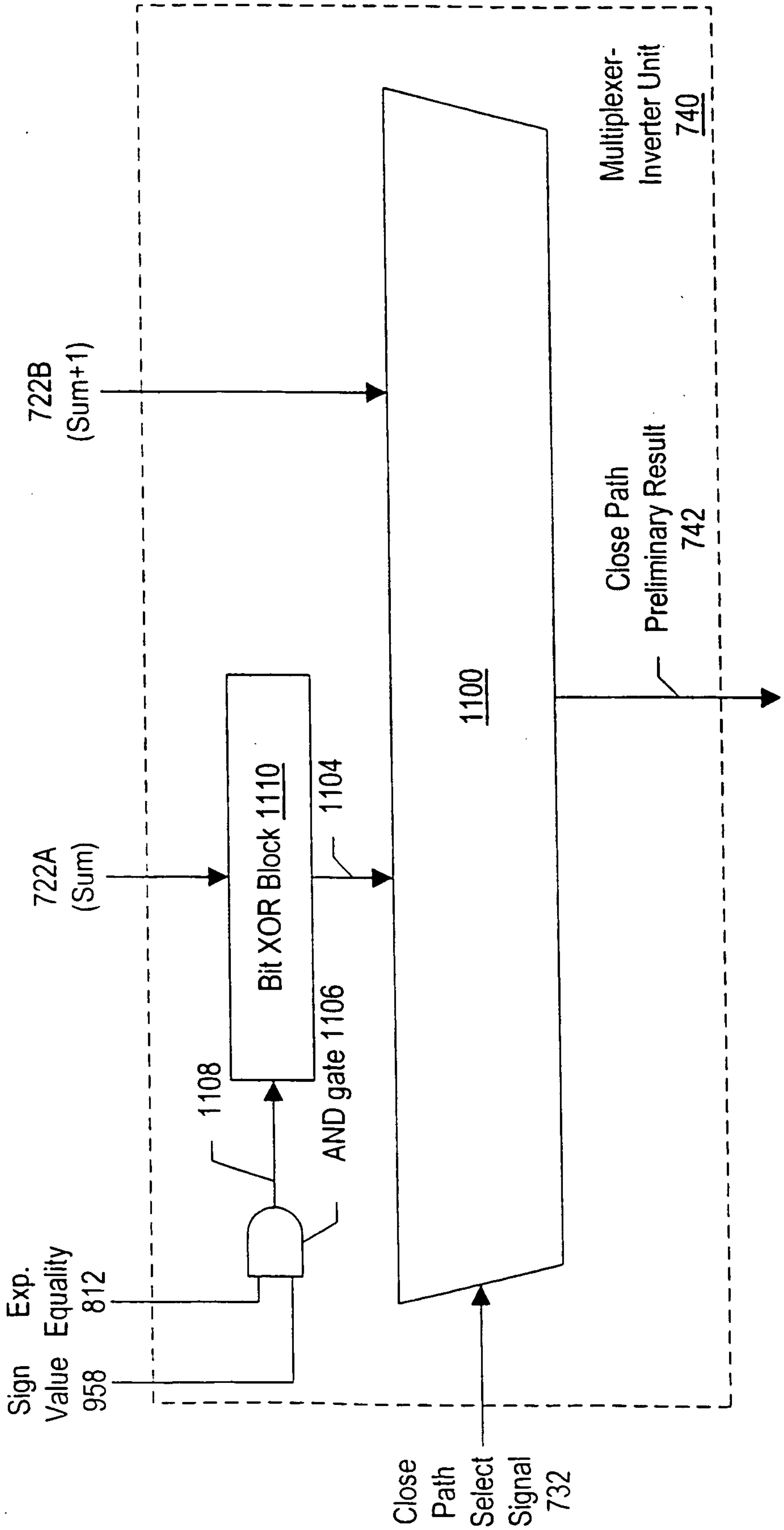

Multiplexer-Inverter Unit 740
722B (Sum+1)
722A (Sum)
Bit XOR Block 1110
1104
1100
Close Path Preliminary Result 742
1108
AND gate 1106
Exp. Equality 812
Sign Value 958
Close Path Select Signal 732

FIG. 17

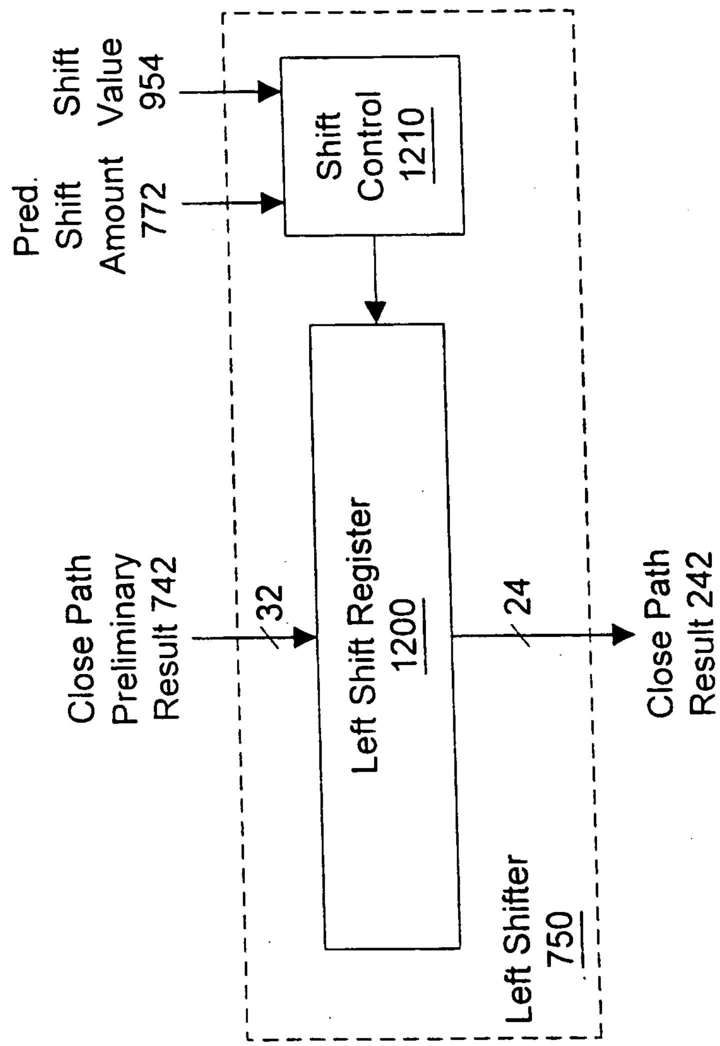

Pred. Shift Amount 772
Shift Value 954
Shift Control 1210
Close Path Preliminary Result 742
32
Left Shift Register 1200
24
Close Path Result 242
Left Shifter 750

FIG. 18

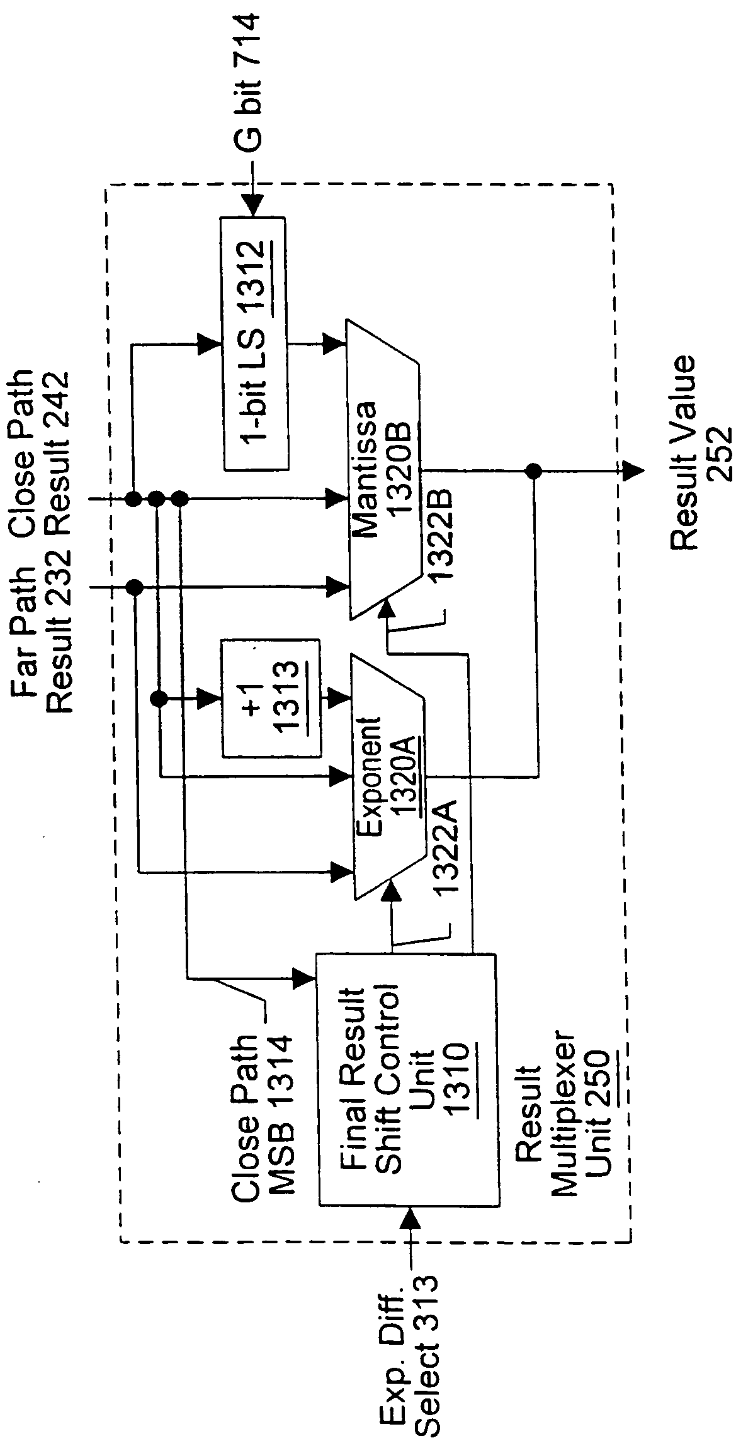

G bit 714
1-bit LS 1312
Far Path Result 232
Close Path Result 242
Mantissa 1320B
1322B
Result Value 252
+1 1313
Exponent 1320A
1322A
Close Path MSB 1314
Final Result Shift Control Unit 1310
Result Multiplexer Unit 250
Exp. Diff. Select 313

FIG. 19

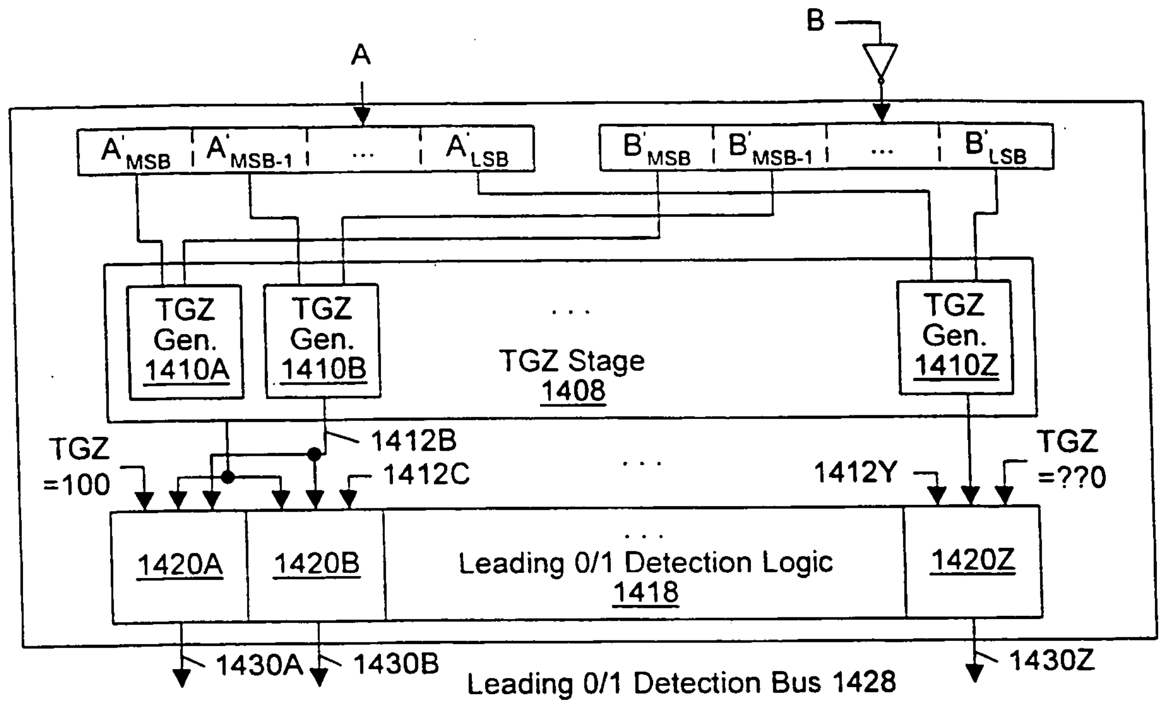


FIG. 20
(PRIOR ART)

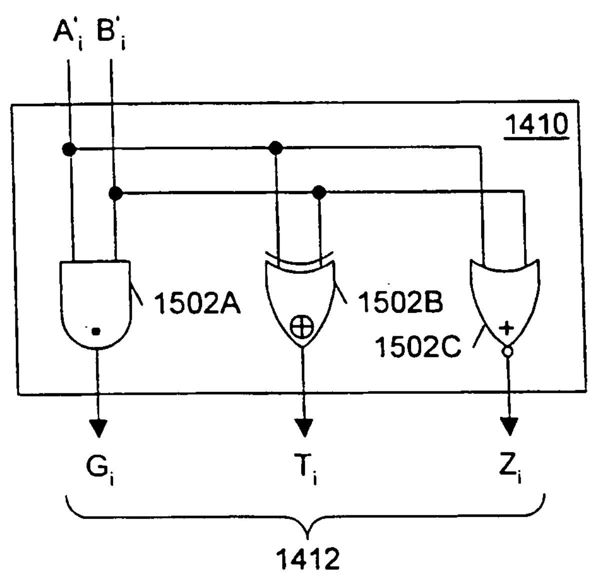


FIG. 21
(PRIOR ART)

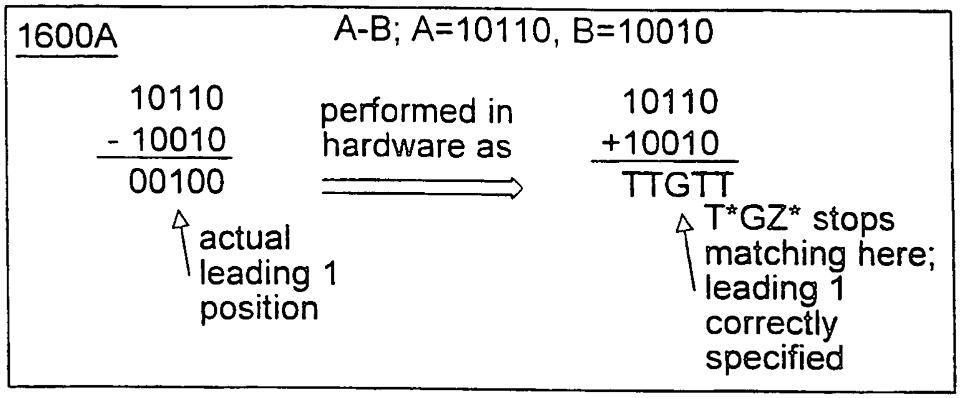


FIG. 22A

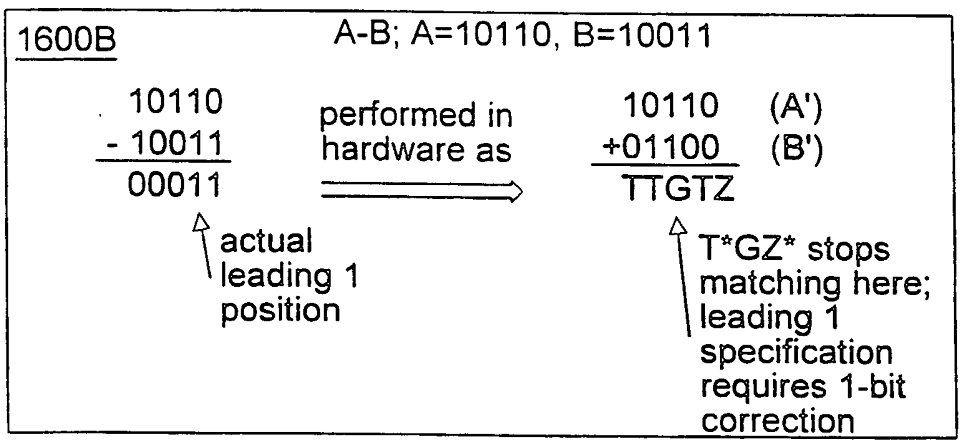


FIG. 22B

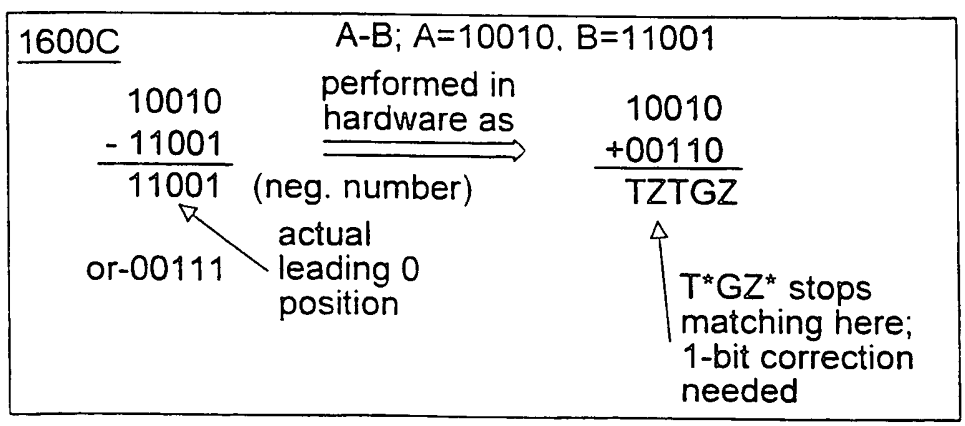


FIG. 22C

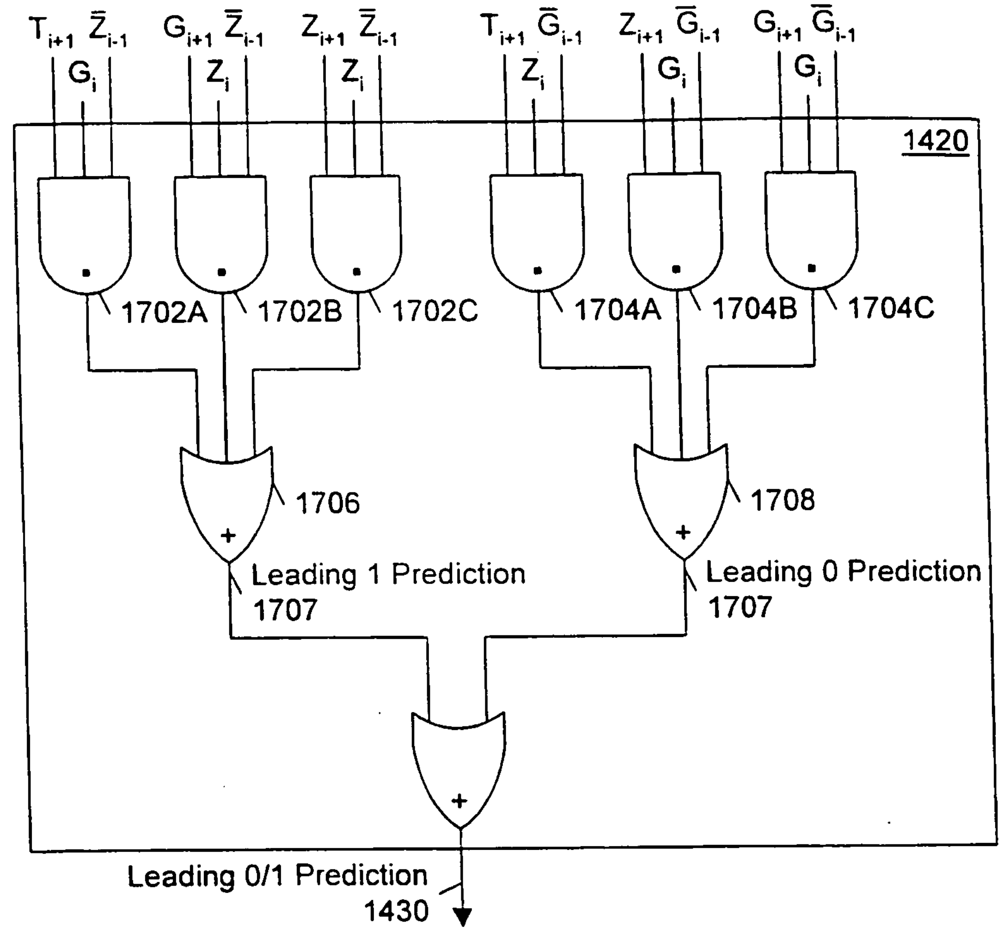


FIG. 23

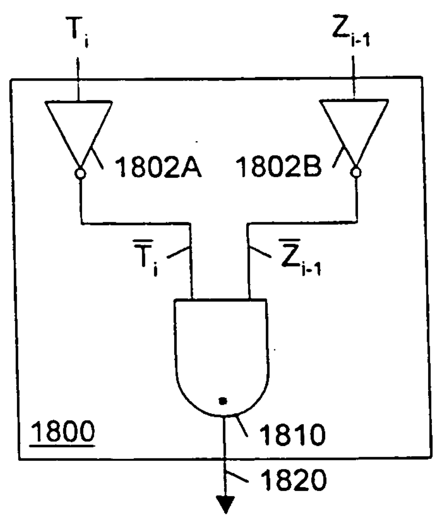


FIG. 24
(PRIOR ART)

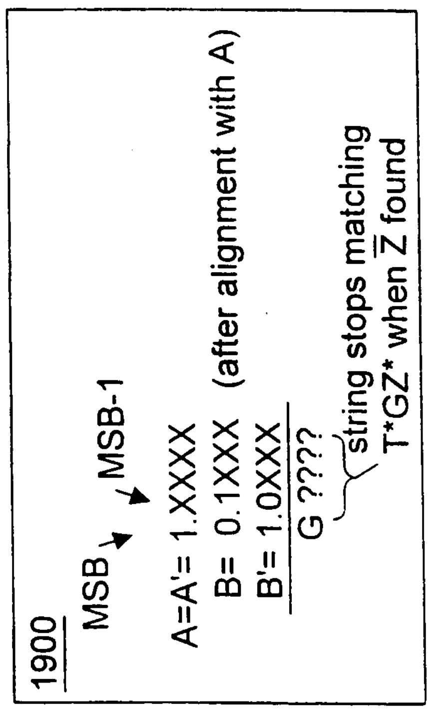
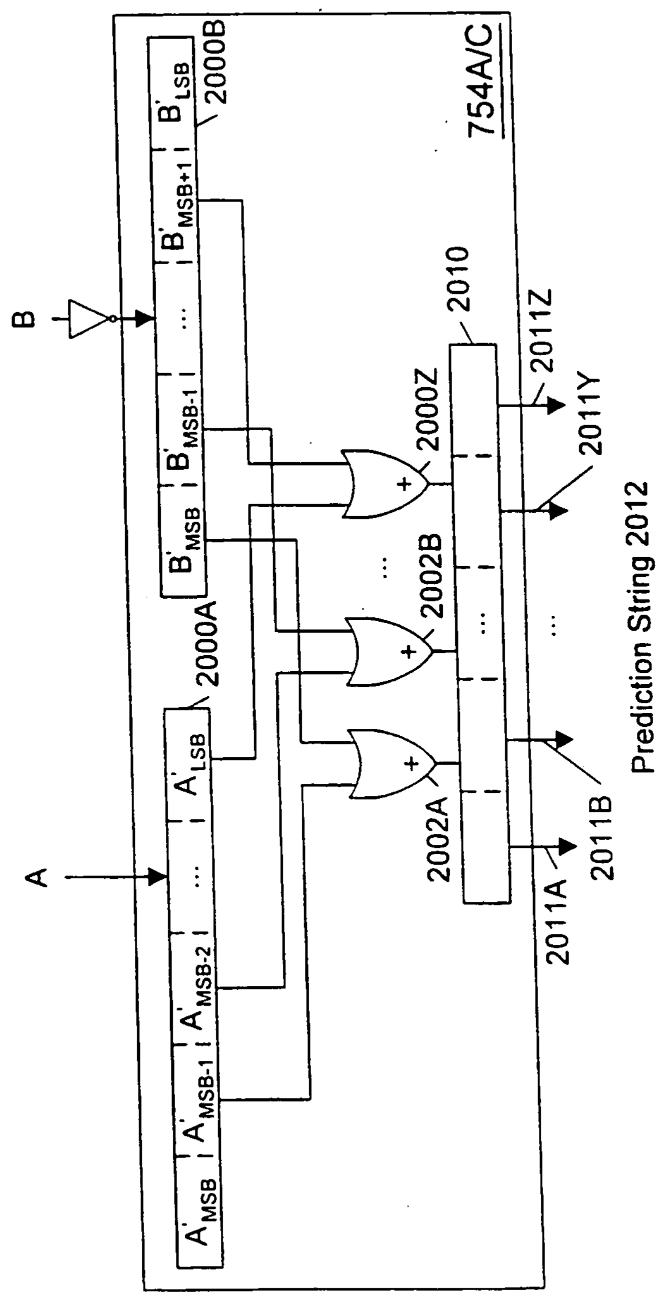

1900
MSB
MSB-1
A=A'= 1.XXXX
B= 0.1XXX (after alignment with A)
B'= 1.0XXX
G ????
string stops matching
T*GZ* when $\overline{Z}$ found

FIG. 25



B
A
B'_{LSB}
B'_{MSB+1}
...
B'_{MSB-1}
B'_{MSB}
2000B
754A/C
A'_{LSB}
...
A'_{MSB-2}
A'_{MSB-1}
A'_{MSB}
2000A
2010
2011Z
2011Y
2000Z
2002B
2002A
2011A
2011B
Prediction String 2012

FIG. 26

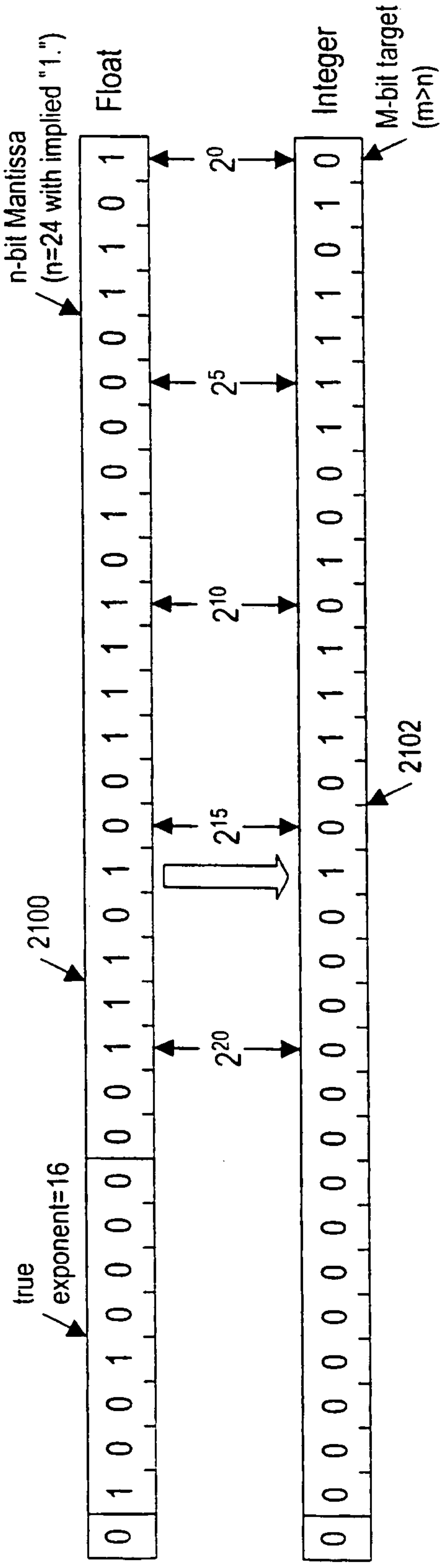


FIG. 27A

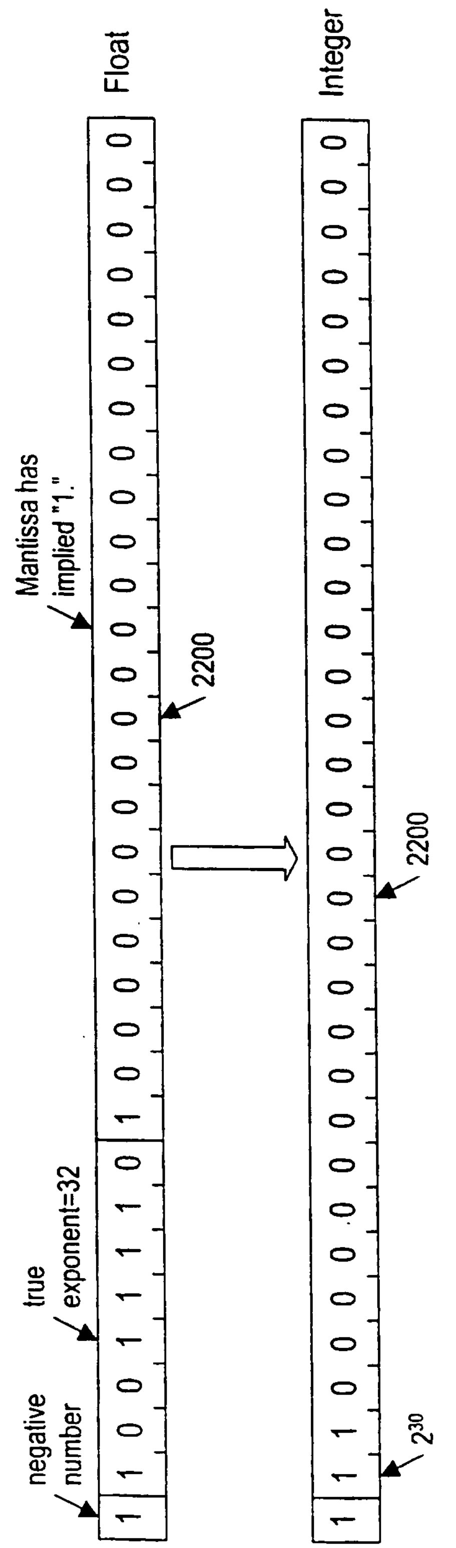
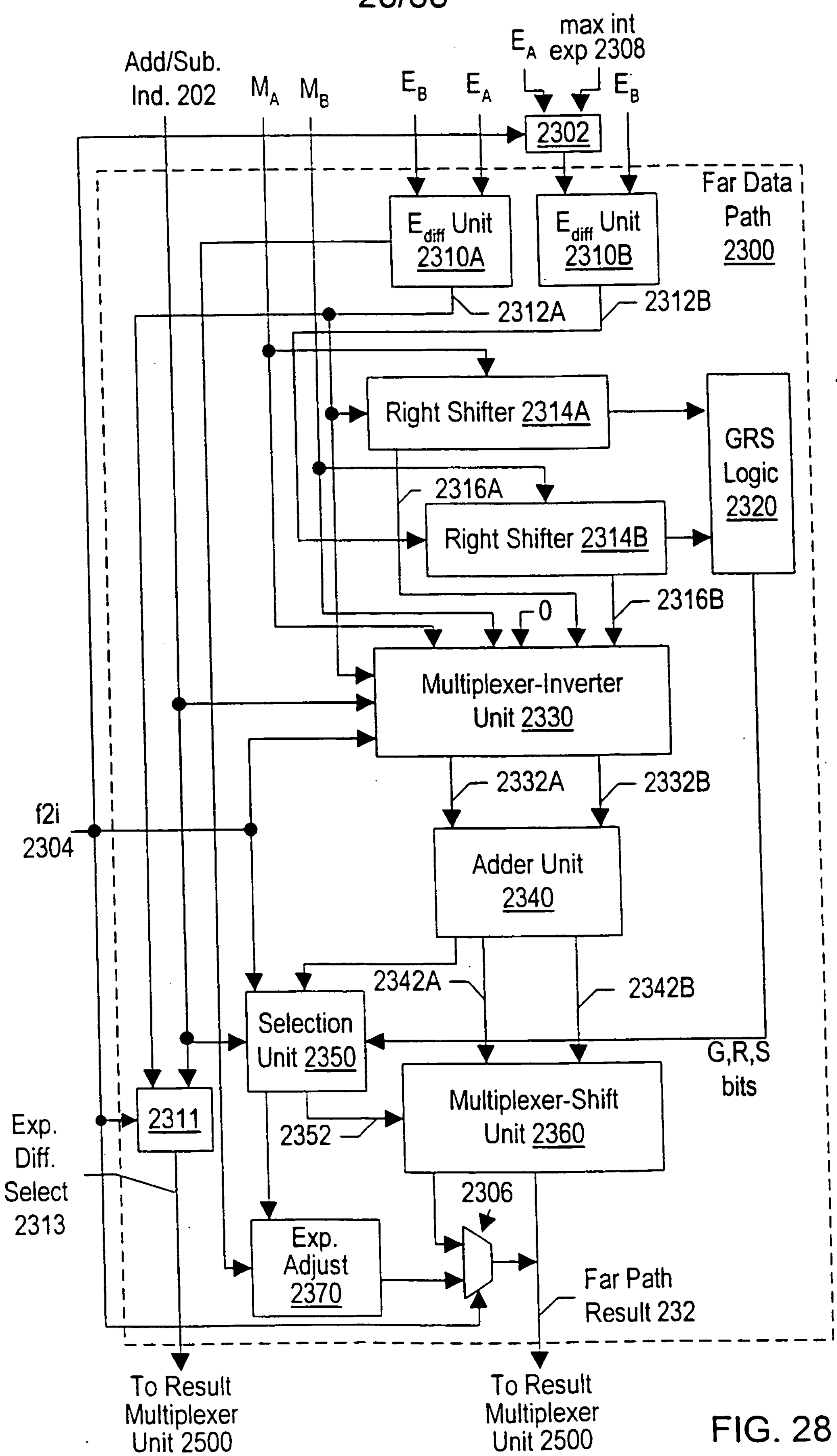


FIG. 27B



Add/Sub. Ind. 202
M_A
M_B
E_B
E_A
E_A
max int exp 2308
E_B
2302
Far Data Path 2300
E_{diff} Unit 2310A
E_{diff} Unit 2310B
2312A
2312B
Right Shifter 2314A
GRS Logic 2320
2316A
Right Shifter 2314B
2316B
0
Multiplexer-Inverter Unit 2330
f2i 2304
2332A
2332B
Adder Unit 2340
2342A
2342B
Selection Unit 2350
G,R,S bits
2311
Exp. Diff. Select 2313
2352
Multiplexer-Shift Unit 2360
2306
Exp. Adjust 2370
Far Path Result 232
To Result Multiplexer Unit 2500
To Result Multiplexer Unit 2500

FIG. 28

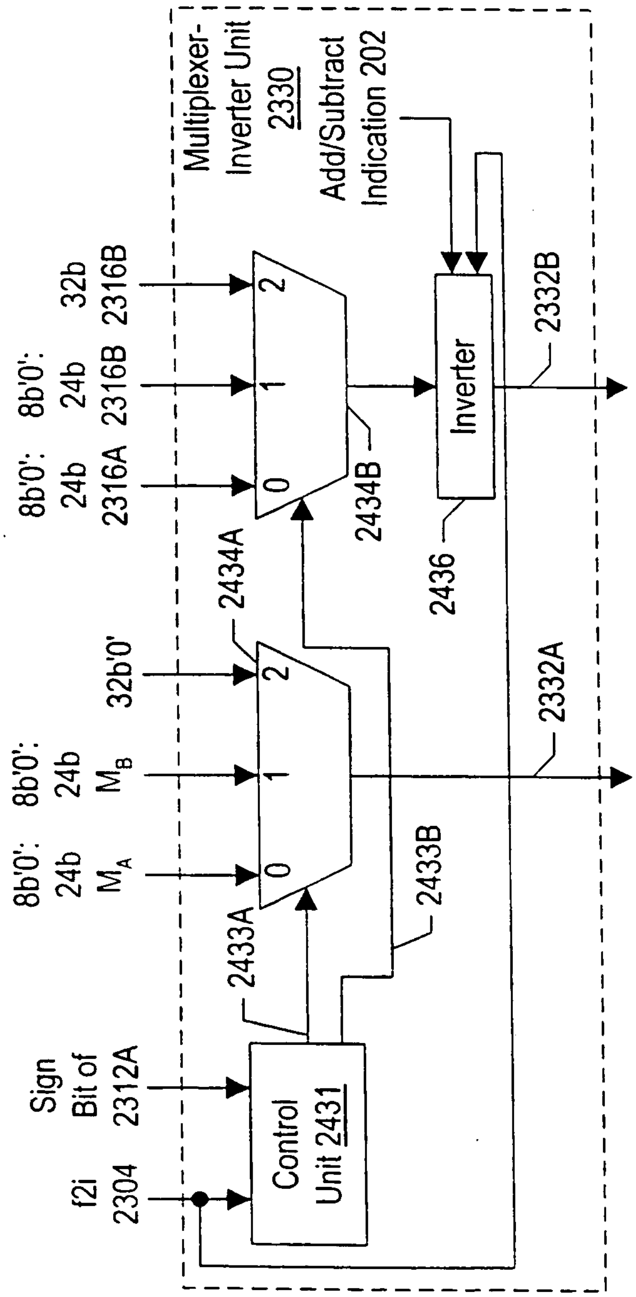

Multiplexer-Inverter Unit 2330
Add/Subtract Indication 202
32b 2316B
8b'0': 24b 2316B
8b'0': 24b 2316A
32b'0'
8b'0': 24b M_B
8b'0': 24b M_A
Sign Bit of 2312A
f2i 2304
Control Unit 2431
Inverter
2
1
0
2434A
2434B
2436
2332B
2332A
2433A
2433B

FIG. 29

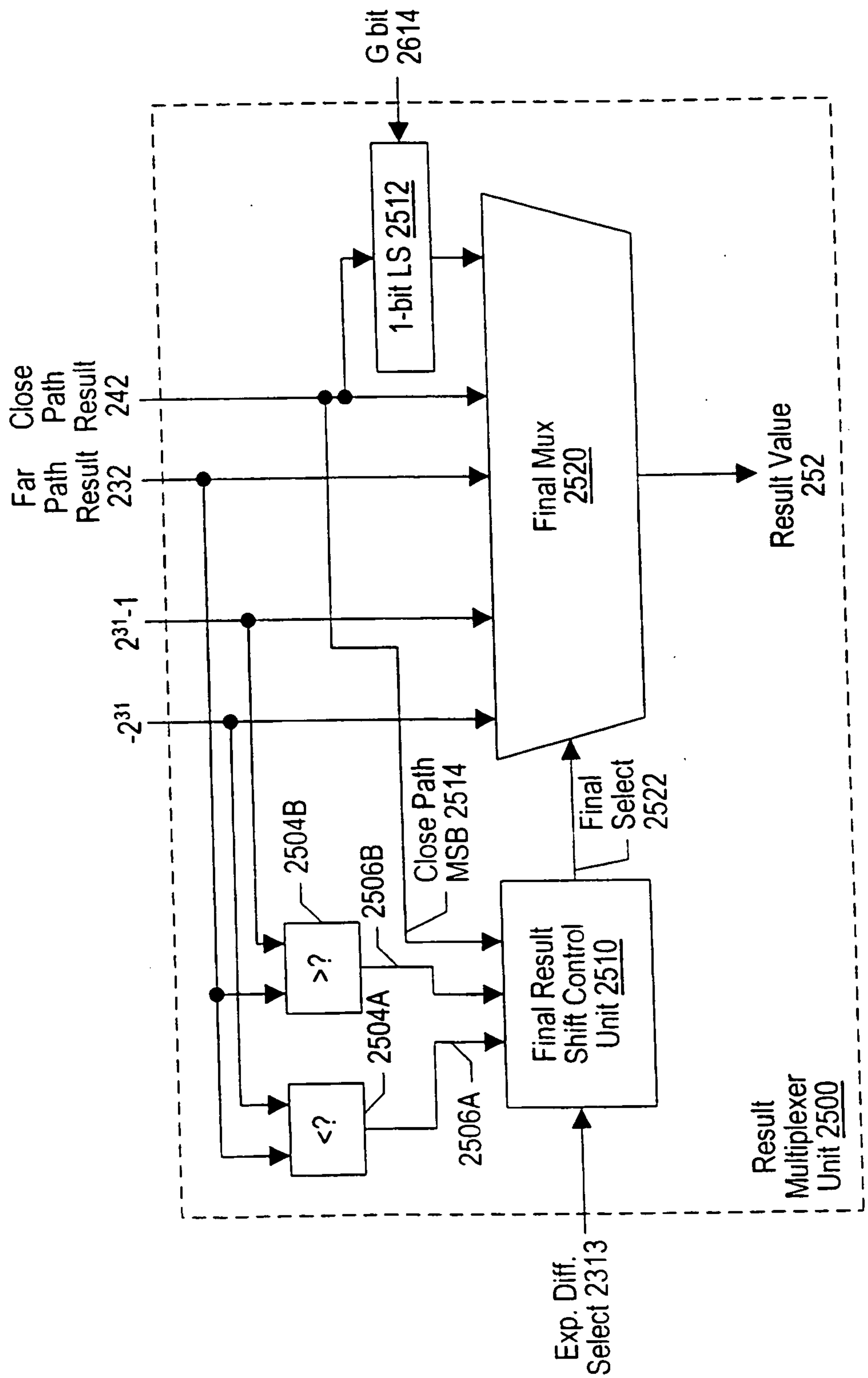

G bit
2614
1-bit LS 2512
Close
Path
Result
242
Far
Path
Result
232
2^{31}-1
-2^{31}
Final Mux
2520
Result Value
252
Final
Select
2522
Close Path
MSB 2514
2504B
2506B
>?
<?
2504A
2506A
Final Result
Shift Control
Unit 2510
Result
Multiplexer
Unit 2500
Exp. Diff.
Select 2313

FIG. 30

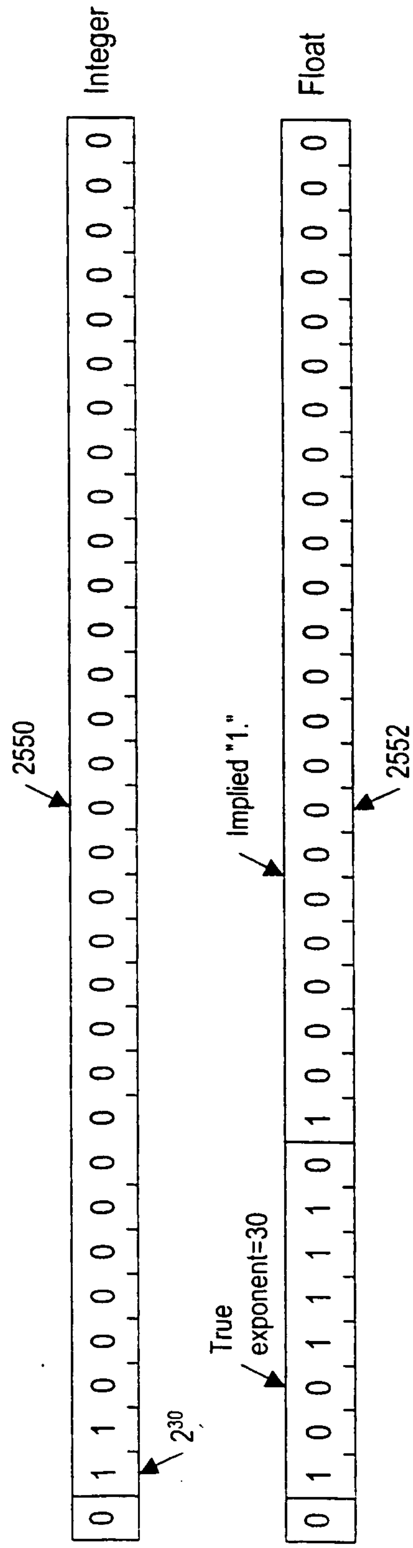


FIG. 31A

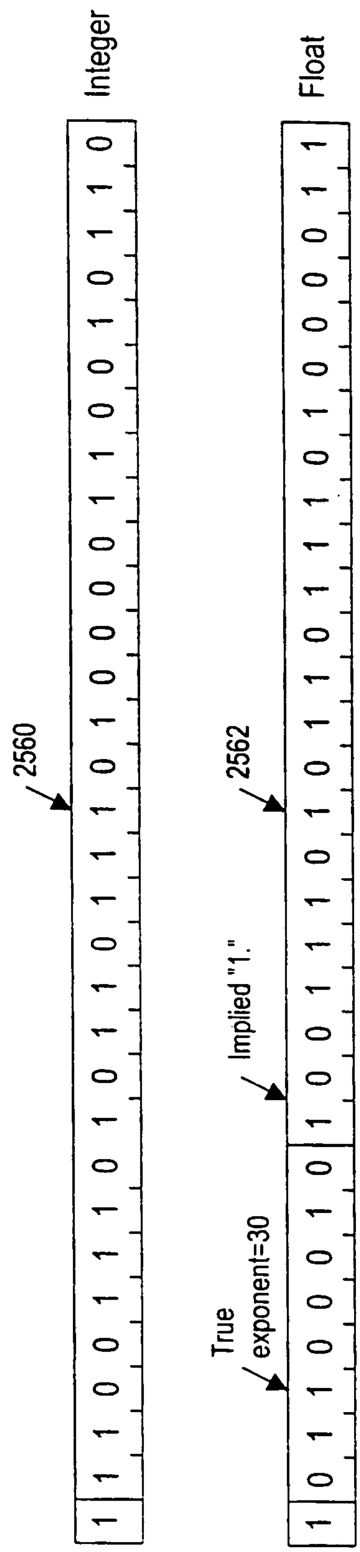


FIG. 31B

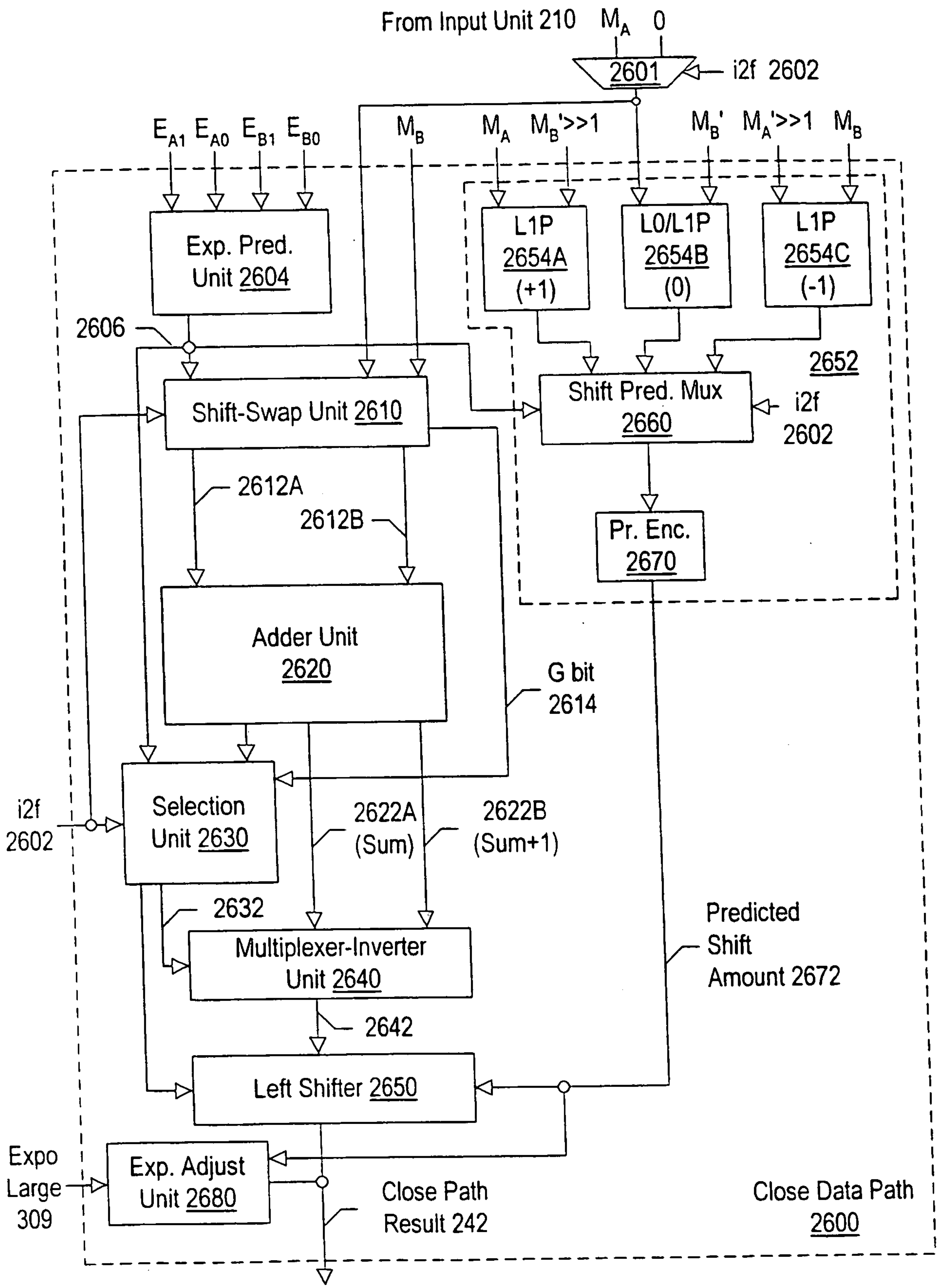

From Input Unit 210
M_A
0
2601
i2f 2602
E_{A1}
E_{A0}
E_{B1}
E_{B0}
M_B
M_A
M_B'>>1
M_B'
M_A'>>1
M_B
Exp. Pred. Unit 2604
L1P 2654A (+1)
L0/L1P 2654B (0)
L1P 2654C (-1)
2606
2652
Shift-Swap Unit 2610
Shift Pred. Mux 2660
i2f 2602
2612A
2612B
Pr. Enc. 2670
Adder Unit 2620
G bit 2614
i2f 2602
Selection Unit 2630
2622A (Sum)
2622B (Sum+1)
2632
Predicted Shift Amount 2672
Multiplexer-Inverter Unit 2640
2642
Left Shifter 2650
Expo Large 309
Exp. Adjust Unit 2680
Close Path Result 242
Close Data Path 2600

FIG. 32

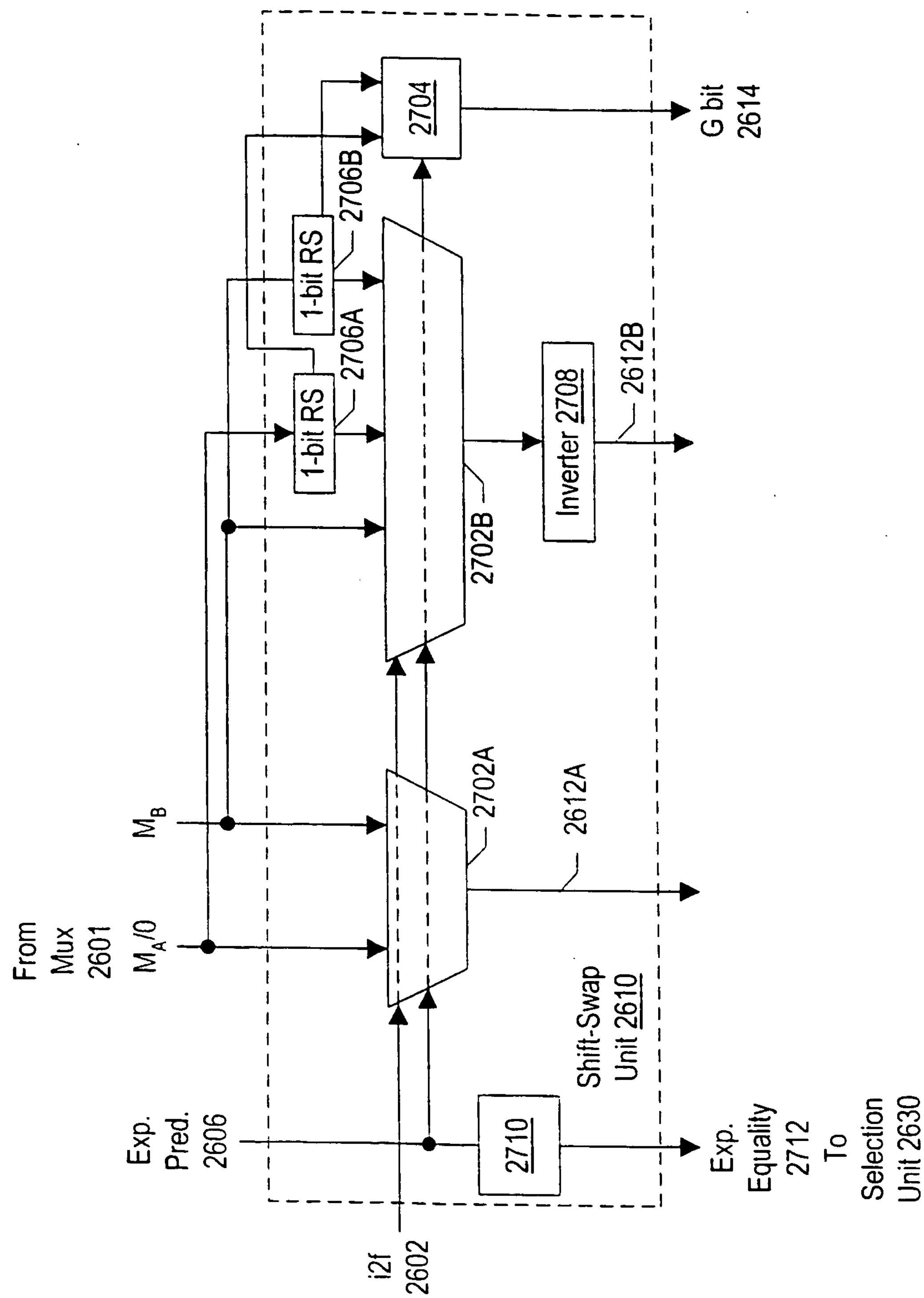

From
Mux
2601
M_A/0
M_B
Exp.
Pred.
2606
i2f
2602
2710
Shift-Swap
Unit 2610
Exp.
Equality
2712
To
Selection
Unit 2630
2702A
2612A
2702B
Inverter 2708
2612B
1-bit RS
2706A
1-bit RS
2706B
2704
G bit
2614

FIG. 33

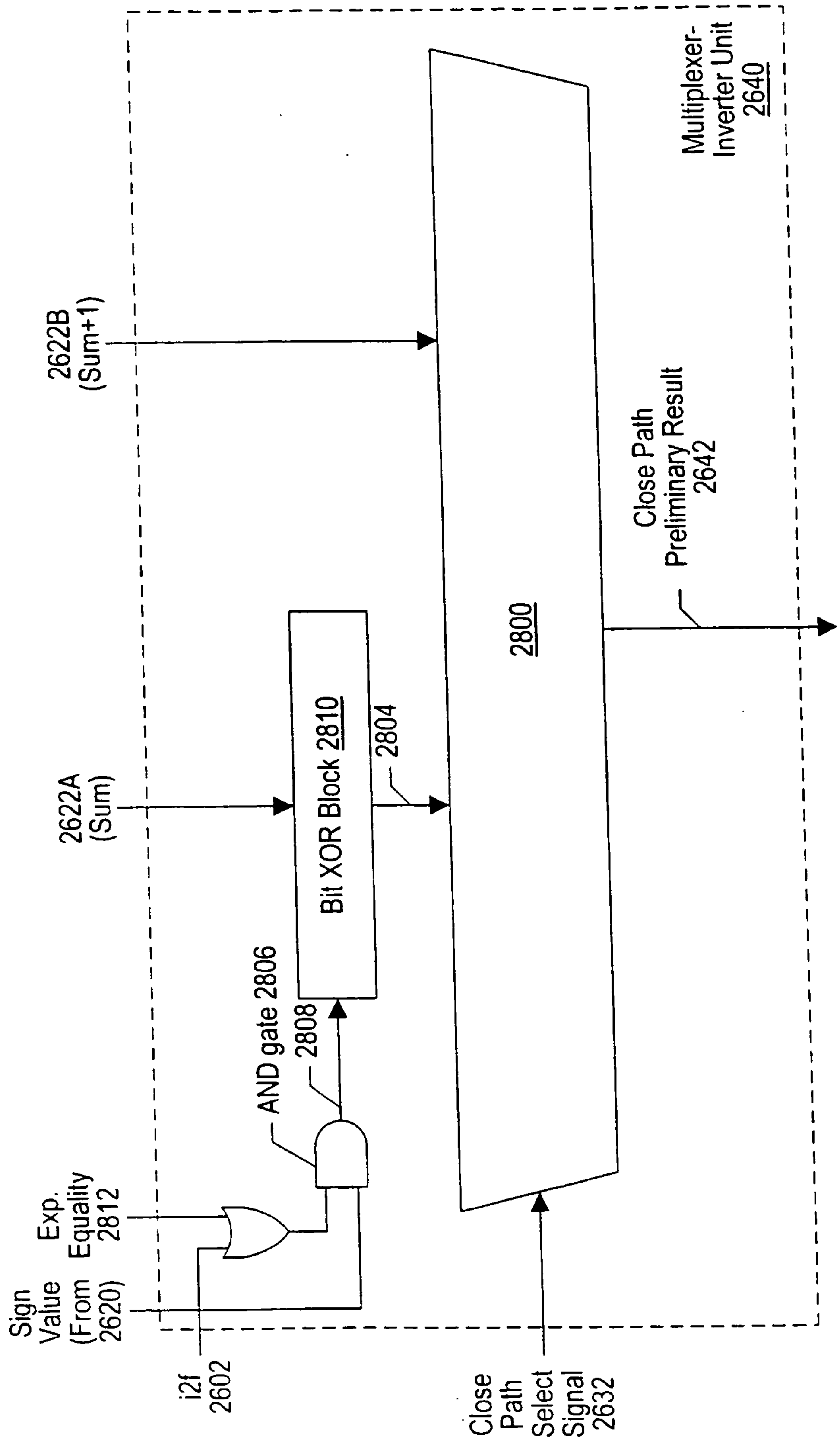

Multiplexer-
Inverter Unit
2640
2622B
(Sum+1)
2622A
(Sum)
Bit XOR Block 2810
2804
2800
Close Path
Preliminary Result
2642
AND gate 2806
2808
Exp.
Equality
2812
Sign
Value
(From
2620)
i2f
2602
Close
Path
Select
Signal
2632

FIG. 34

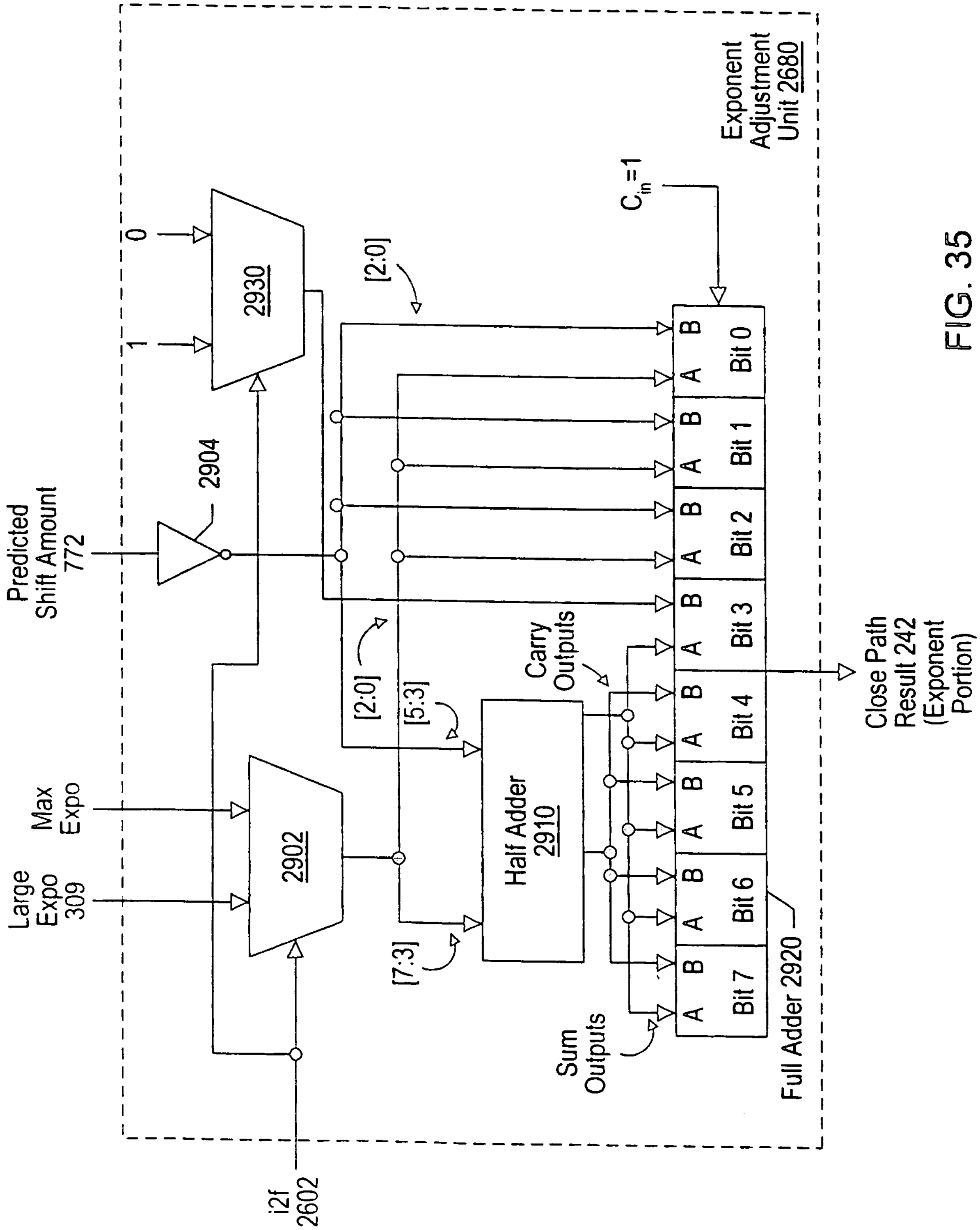

Exponent
Adjustment
Unit 2680
$C_{in}=1$
0
1
2930
[2:0]
Predicted
Shift Amount
772
2904
B
Bit 0
A
B
Bit 1
A
B
Bit 2
A
B
Bit 3
A
B
Bit 4
A
B
Bit 5
A
B
Bit 6
A
B
Bit 7
A
[2:0]
Carry
Outputs
Close Path
Result 242
(Exponent
Portion)
Max
Expo
Large
Expo
309
2902
[5:3]
Half Adder
2910
[7:3]
Sum
Outputs
Full Adder 2920
i2f
2602

FIG. 35

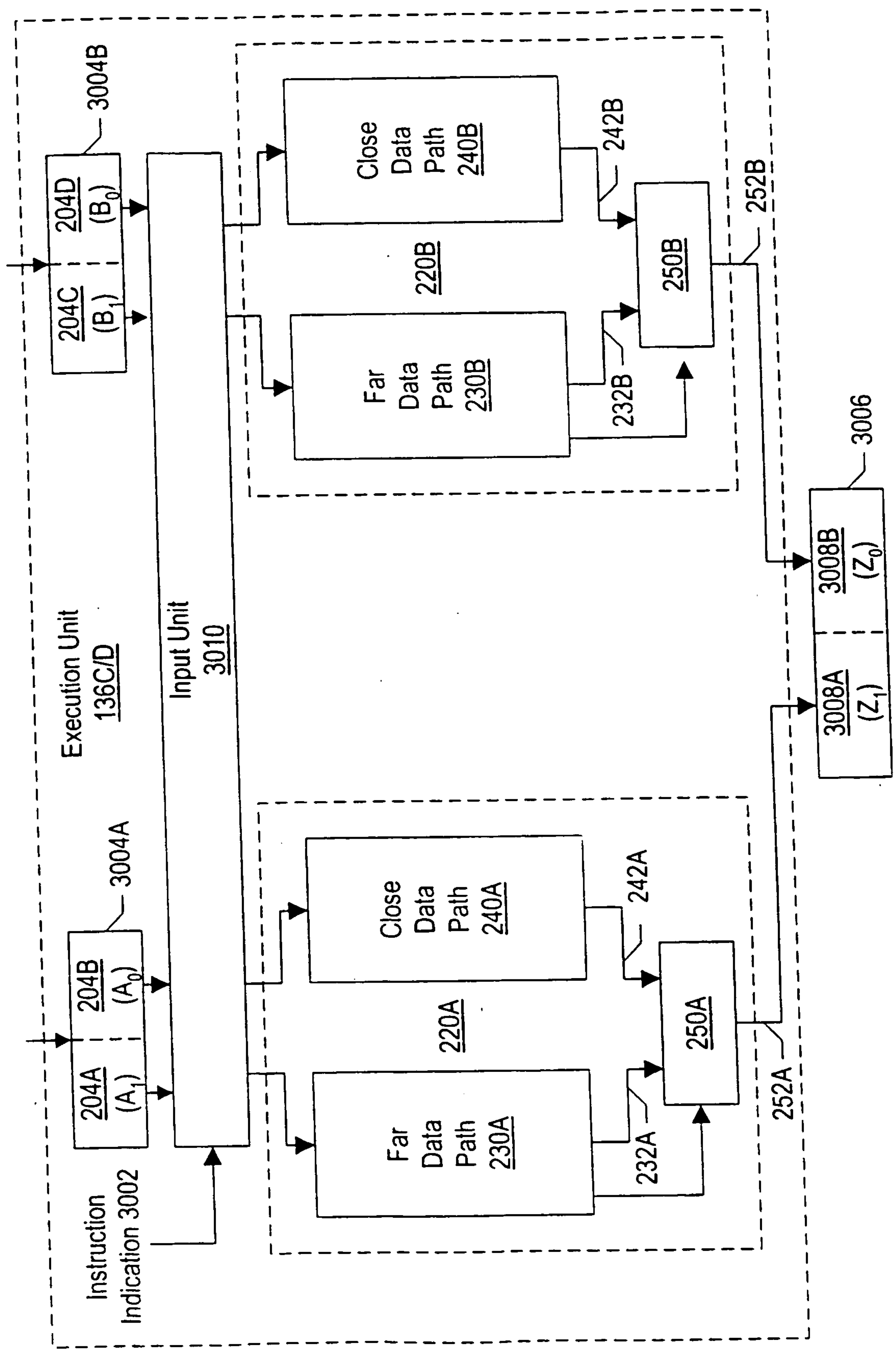

Execution Unit
136C/D
3004B
204D
(B_0)
204C
(B_1)
Input Unit
3010
Close
Data
Path
240B
242B
250B
220B
Far
Data
Path
230B
232B
252B
3006
3008B
(Z_0)
3008A
(Z_1)
3004A
204B
(A_0)
204A
(A_1)
Close
Data
Path
240A
242A
250A
220A
Far
Data
Path
230A
232A
252A
Instruction
Indication 3002

FIG. 36

PFADD

3100

mnemonic	opcode/imm8	description
PFADD mmreg1,mmreg2/mem64	0Fh 0Fh / 9Eh	Packed floating-point addition

3102A 3102B 3101

FIG. 37A

mmreg1[31:0] = mmreg1[31:0] + mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] + mmreg2/mem64[63:32]

3104

FIG. 37B

PFSUB

3110

mnemonic	opcode/imm8	description
PFSUB mmreg1,mmreg2/mem64	0Fh 0Fh / 9Ah	Packed floating-point subtraction

3112A 3112B 3111

FIG. 38A

mmreg1[31:0] = mmreg1[31:0] - mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] - mmreg2/mem64[63:32]

3114

FIG. 38B

PF2ID

3120

mnemonic	opcode/imm8	description
PF2ID mmreg1,mmreg2/mem64	0Fh 0Fh / 1Dh	Converts packed floating-point to packed 32-bit integer

3122A 3122B 3121

FIG. 39A

```
IF (mmreg2/mem64[31:0] >=2^31)
        THEN mmreg1[31:0] =7FFF_FFFFh
ELSEIF (mmreg2/mem64[31:0] <=-2^31)
        THEN mmreg1[31:0] = 8000_0000h
ELSE mmreg1[31:0] = int(mmreg2/mem64[31:0])
IF (mmreg2/mem64[63:32] >=2^31)
        THEN mmreg1[63:32] = 7FFF_FFFFh
ELSEIF (mmreg2/mem64[63:32] <=-2^31)
        THEN mmreg1[63:32] = 8000_0000h
ELSE mmreg1[63:32] = int(mmreg2/mem64[63:32])
```

3124

FIG. 39B

PF2ID		
Source 1 & Destination	0	0
	Normal, abs(Source 1) <1	0
	Normal, -4294967297 < Source 1 <=-1	round to zero (Source 1)
	Normal, 1 <= Source 1< 4294967296	round to zero (Source 1)
	Normal, Source 1 >= 4294967296	7FFF_FFFFh
	Normal, Source 1 <= -4294967297	8000_0000h
	Unsupported	Undefined

3128

FIG. 39C

PF2IW

3130

mnemonic	opcode/imm8	description
PF2IW mmreg1,mmreg2/mem64	0Fh 0Fh / 1Ch	Converts packed floating-point to packed 16-bit integer

3132A 3132B 3131

FIG. 40A

```
IF (mmreg2/mem64[31:0] >=2^15)
    THEN mmreg1[15:0] =7FFFh
ELSEIF (mmreg2/mem64[31:0] <=-2^15)
    THEN mmreg1[15:0] = 8000h
ELSE mmreg1[15:0] = int(mmreg2/mem64[31:0])
IF (mmreg2/mem64[63:32] >=2^15)
    THEN mmreg1[47:32] = 7FFFh
ELSEIF (mmreg2/mem64[63:32] <=-2^15)
    THEN mmreg1[47:32] = 8000h
ELSE mmreg1[47:32] = int(mmreg2/mem64[63:32])
```

3134

FIG. 40B

PF2IW		
Source 1 & Destination	0	0
	Normal, abs(Source 1) <1	0
	Normal, -65537 < Source 1 <=-1	round to zero (Source 1)
	Normal, 1 <= Source 1< 65536	round to zero (Source 1)
	Normal, Source 1 >= 65536	0x7FFFh
	Normal, Source 1 <= -65537	0x8000h
	Unsupported	Undefined

3138

FIG. 40C

PI2FD

3140

mnemonic	opcode/imm8	description
PI2FD mmreg1,mmreg2/mem64	0Fn 0Fn / 0Dh	Packed 32-bit integer to floating-point conversion

3142A 3142B 3141

FIG. 41A

mmreg1[31:0] = float(mmreg2/mem64[31:0]
mmreg1[63:32] = float(mmreg2/mem64[63:32])

3144

FIG. 41B

PI2FW

3150

mnemonic	opcode/imm8	description
PI2FW mmreg1,mmreg2/mem64	0Fn 0Fn / 0Ch	Packed 16-bit integer to floating-point conversion

3152A 3152B 3151

FIG. 42A

mmreg1[31:0] = float(mmreg2/mem64[15:0]
mmreg1[63:32] = float(mmreg2/mem64[47:32])

3154

FIG. 42B

PFACC

3160

mnemonic	opcode/imm8	description
PFACC mmreg1,mmreg2/mem64	0Fh 0Fh / AEh	Floating-point accumulate

3162A 3162B 3161

FIG. 43A

mmreg1[31:0] = mmreg1[31:0] + mmreg1[63:32]
mmreg1[63:32] = mmreg2/mem64[31:0] + mmreg2/mem64[63:32]

3164

FIG. 43B

PFSUBR

3170

mnemonic	opcode/imm8	description
PFSUBR mmreg1,mmreg2/mem64	0Fh 0Fh / AAh	Packed floating-point reverse subtraction

3172A 3172B 3171

FIG. 44A

mmreg1[31:0] = mmreg2/mem64[31:0] - memreg1[31:0]
mmreg1[63:32] = mmreg2/mem64[63:32] - memreg1[63:32]

3174

FIG. 44B

PFMAX

3180

mnemonic	opcode/imm8	description
PFMAX mmreg1,mmreg2/mem64	0Fh 0Fh / A4h	Packed floating-point maximum

3182A 3182B 3181

FIG. 45A

```
IF (memreg1[31:0] > memreg2/mem64[31:0]
       THEN memreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] > mmreg2/mem64[63:32])
       THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]
```

3184

FIG. 45B

PFMAX	Source 2			
		0	Normal	Unsupported
Source 1 & Destination	0	+0	Source 2, +0**	Undefined
	Normal	Source 1, +0**	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:
* The result is source 2 if source 2 is positive otherwise the result is positive zero.
** The result is source 1 if source 1 is positive otherwise the result is positive zero.
*** The result is source 1 if source 1 is positive and source 2 is negative. The result is source 1 if both are positive and source 1 is greater in magnitude than source 2. The result is source 1 if both are negative and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.

3188

FIG. 45C

PFMIN

3190

mnemonic	opcode/imm8	description
PFMIN mmreg1,mmreg2/mem64	0Fh 0Fh / 94h	Packed floating-point maximum

3192A 3192B 3191

FIG. 46A

```
IF (memreg1[31:0] < memreg2/mem64[31:0])
        THEN memreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] < mmreg2/mem64[63:32])
        THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]
```

3194

FIG. 46B

PFMIN		Source 2		
Source 1 & Destination		0	Normal	Unsupported
	0	+0	Source 2, +0*	Undefined
	Normal	Source 1, +0**	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:
* The result is source 2 if source 2 is negative otherwise the result is positive zero.
** The result is source 1 if source 1 is negative otherwise the result is positive zero.
*** The result is source 1 if source 1 is negative and source 2 is positive. The result is source 1 if both are negative and source 1 is greater in magnitude than source 2. The result is source 1 if both are positive and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.

3198

FIG. 46C

PFCMPEQ

3200

mnemonic	opcode/imm8	description
PFCMPEQ mmreg1,mmreg2/mem64	0Fh 0Fh / B0h	Packed floating-point comparison, equal

3202A 3202B 3201

FIG. 47A

```
IF (memreg1[31:0] = memreg2/mem64[31:0])
        THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] = mmreg2/mem64[63:32])
        THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

3204

FIG. 47B

PFCMPEQ		Source 2		
		0	Normal	Unsupported
Source 1 & Destination	0	FFFF_FFFFh*	0000_0000h	0000_0000h
	Normal	0000_0000h	0000_0000h, FFFF_FFFFh**	0000_0000h
	Unsupported	0000_0000h	0000_0000h	Undefined

Notes:
* Positive zero is equal to negative zero.
** The result is FFFF_FFFFh if source 1 and source 2 have identical signs, exponents, and mantissas. It is 0000_0000h otherwise.

3208

FIG. 47C

PFCMPGT

3210

mnemonic	opcode/imm8	description
PFCMPGT mmreg1,mmreg2/mem64	0Fh 0Fh / A0h	Packed floating-point comparison, greater

3212A 3212B 3211

FIG. 48A

```
IF (memreg1[31:0] > memreg2/mem64[31:0])
        THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] > mmreg2/mem64[63:32])
        THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

3214

FIG. 48B

PFCMPGT		Source 2		
		0	Normal	Unsupported
Source 1 & Destination	0	0000_0000h	0000_0000h, FFFF_FFFF**	Undefined
	Normal	0000_0000h, FFFF_FFFF**	0000_0000h, FFFF_FFFF***	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:
* The result is FFFF_FFFFh if source 2 is negative, otherwise the result is 0000_0000h.
** The result is FFFF_FFFFh if source 1 is positive, otherwise the result is 0000_0000h.
*** The result is FFFF_FFFFh if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller in magnitude than source 2, or if source 1 and source 2 are positive and source 1 is greater in magnitude than source 2. The result is 0000_0000h in all other cases.

3218

FIG. 48C

PFCMPGE

3220

mnemonic	opcode/imm8	description
PCMPGE mmreg1,mmreg2/mem64	0Fh 0Fh / 90h	Packed floating-point comparison, greater or equal

3222A 3222B 3221

FIG. 49A

```
IF (mmreg1[31:0] >= memreg2/mem64[31:0])
      THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] >= mmreg2/mem64[63:32])
      THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```

3224

FIG. 49B

PFCMPGE		Source 2		
		0	Normal	Unsupported
Source 1 & Destination	0	FFFF_FFFFh	0000_0000h, FFFF_FFFF**	Undefined
	Normal	0000_0000h, FFFF_FFFF***	0000_0000h, FFFF_FFFF****	Undefined
	Unsupported	Undefined	Undefined	Undefined

Notes:
- * Positive zero is equal to negative zero.
- ** The result is FFFF_FFFFh if source 2 is negative, otherwise the result is 0000_0000h.
- *** The result is FFFF_FFFFh if source 1 is positive, otherwise the result is 0000_0000h.
- *** The result is FFFF_FFFFh if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller or equal in magnitude than source 2, or if source 1 and source 2 are both positive and source 1 is greater or equal in magnitude than source 2. The result is 0000_0000h in all other cases.

3228

FIG. 49C

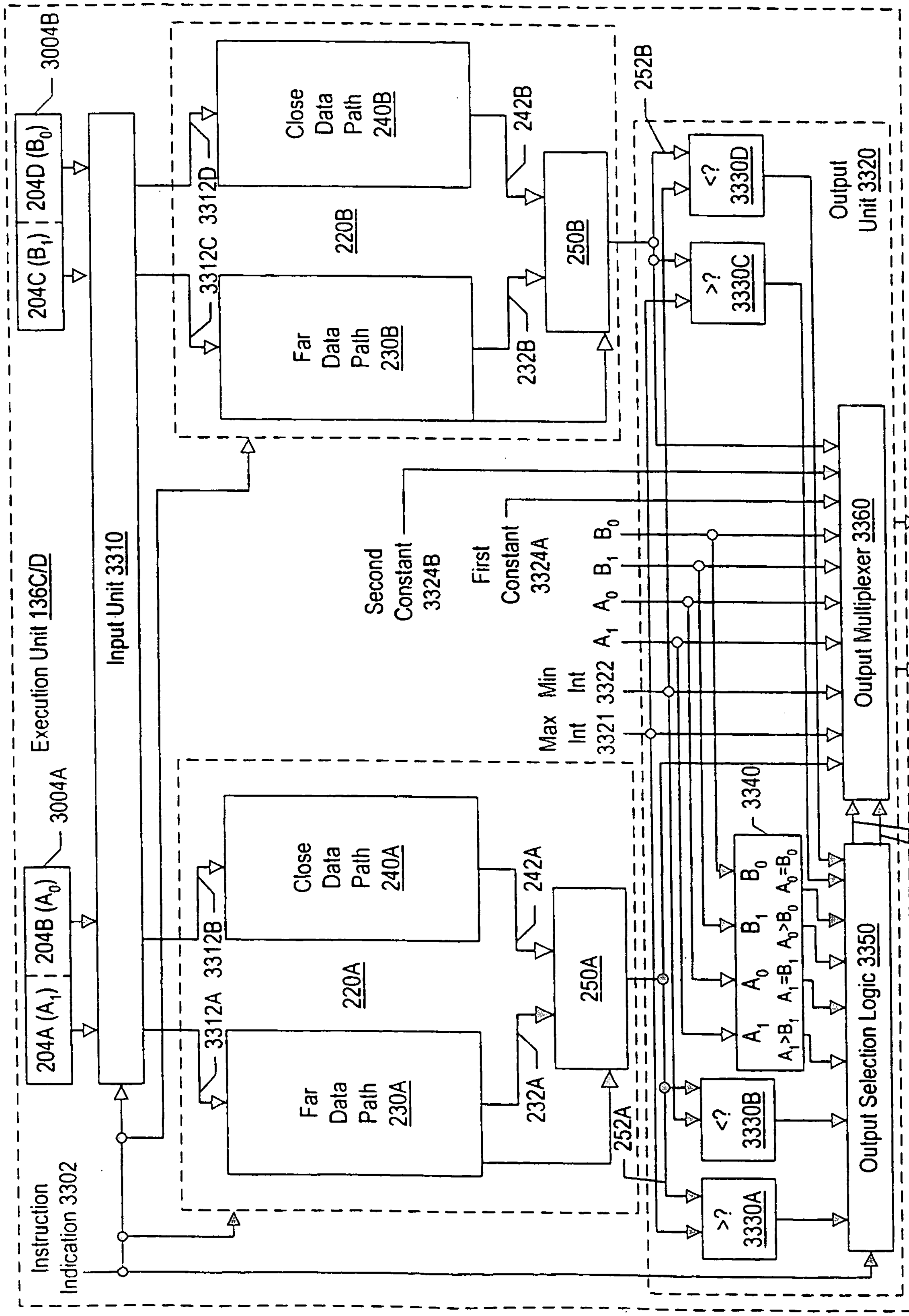

Execution Unit 136C/D
204A (A1)
204B (A0)
3004A
204C (B1)
204D (B0)
3004B
Input Unit 3310
Instruction Indication 3302
3312A
3312B
3312C
3312D
220A
220B
Far Data Path 230A
Close Data Path 240A
Far Data Path 230B
Close Data Path 240B
232A
242A
232B
242B
250A
250B
252A
252B
Second Constant 3324B
First Constant 3324A
Max Int 3321
Min Int 3322
A1
A0
B1
B0
>? 3330A
<? 3330B
>? 3330C
<? 3330D
3340
A1>B1
A1=B1
A0>B0
A0=B0
Output Selection Logic 3350
Output Multiplexer 3360
Output Unit 3320
3352A
3352B
3008A (Z1)
3008B (Z0)
3006

FIG. 50

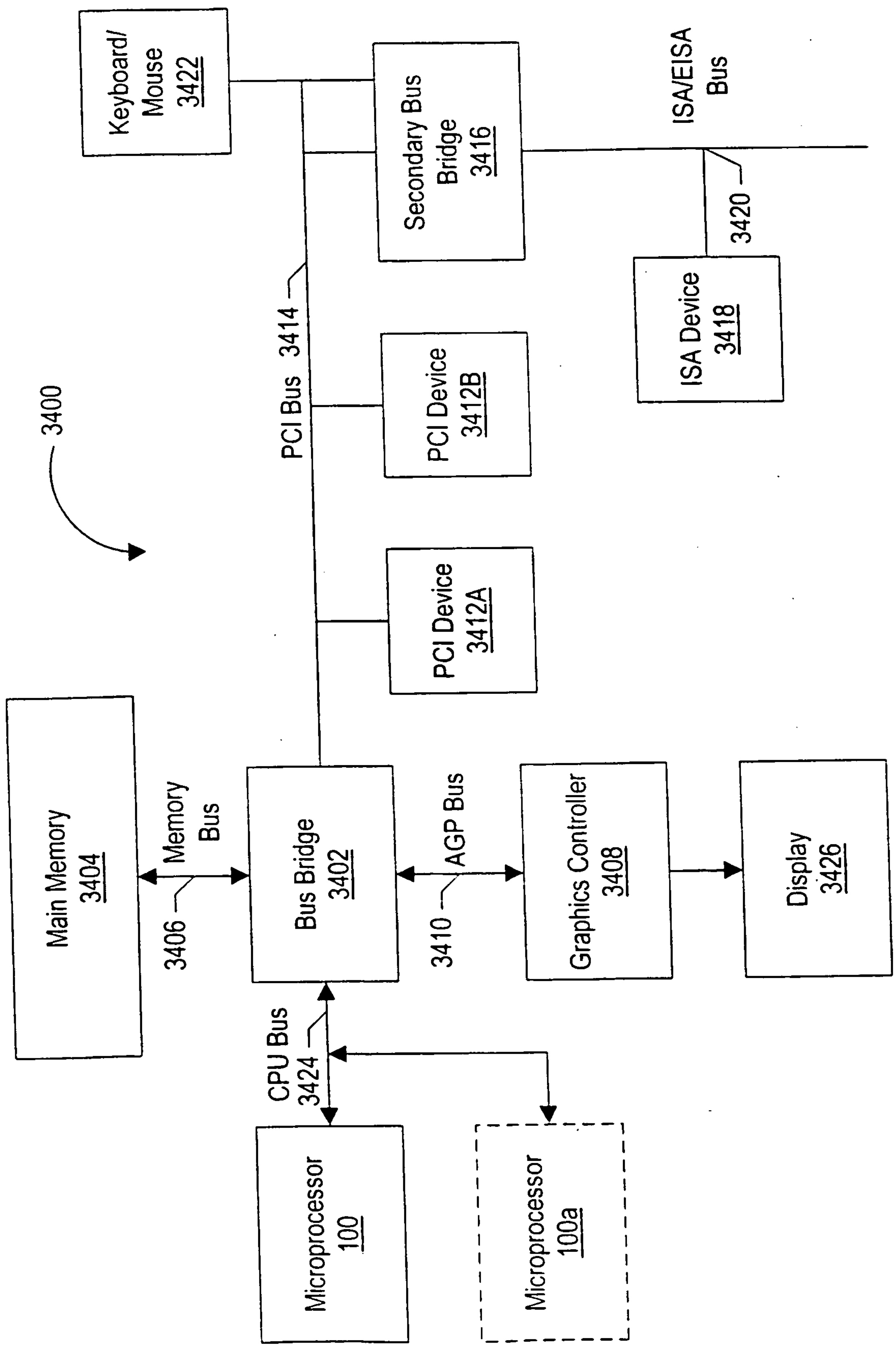

Keyboard/
Mouse
3422
Secondary Bus
Bridge
3416
ISA/EISA
Bus
3420
ISA Device
3418
PCI Bus 3414
3400
PCI Device
3412B
PCI Device
3412A
Main Memory
3404
Memory
Bus
3406
Bus Bridge
3402
AGP Bus
3410
Graphics Controller
3408
Display
3426
CPU Bus
3424
Microprocessor
100
Microprocessor
100a

FIG. 51

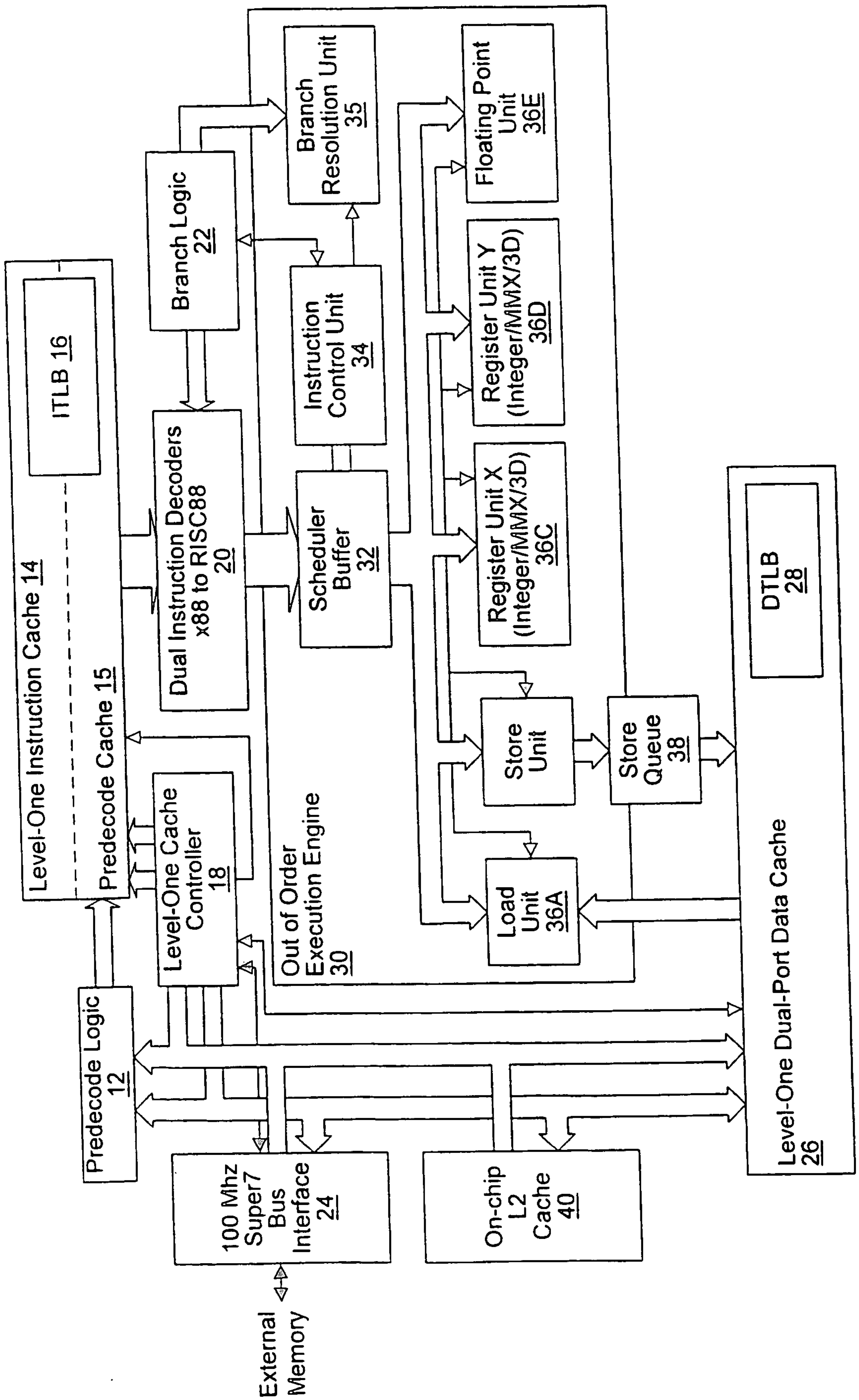

Level-One Instruction Cache 14
ITLB 16
Predecode Cache 15
Branch Logic 22
Branch Resolution Unit 35
Dual Instruction Decoders x88 to RISC88 20
Instruction Control Unit 34
Scheduler Buffer 32
Floating Point Unit 36E
Register Unit Y (Integer/MMX/3D) 36D
Register Unit X (Integer/MMX/3D) 36C
Store Unit
Store Queue 38
Load Unit 36A
Level-One Cache Controller 18
Out of Order Execution Engine 30
DTLB 28
Level-One Dual-Port Data Cache 26
Predecode Logic 12
100 Mhz Super7 Bus Interface 24
On-chip L2 Cache 40
External Memory

FIG. 52

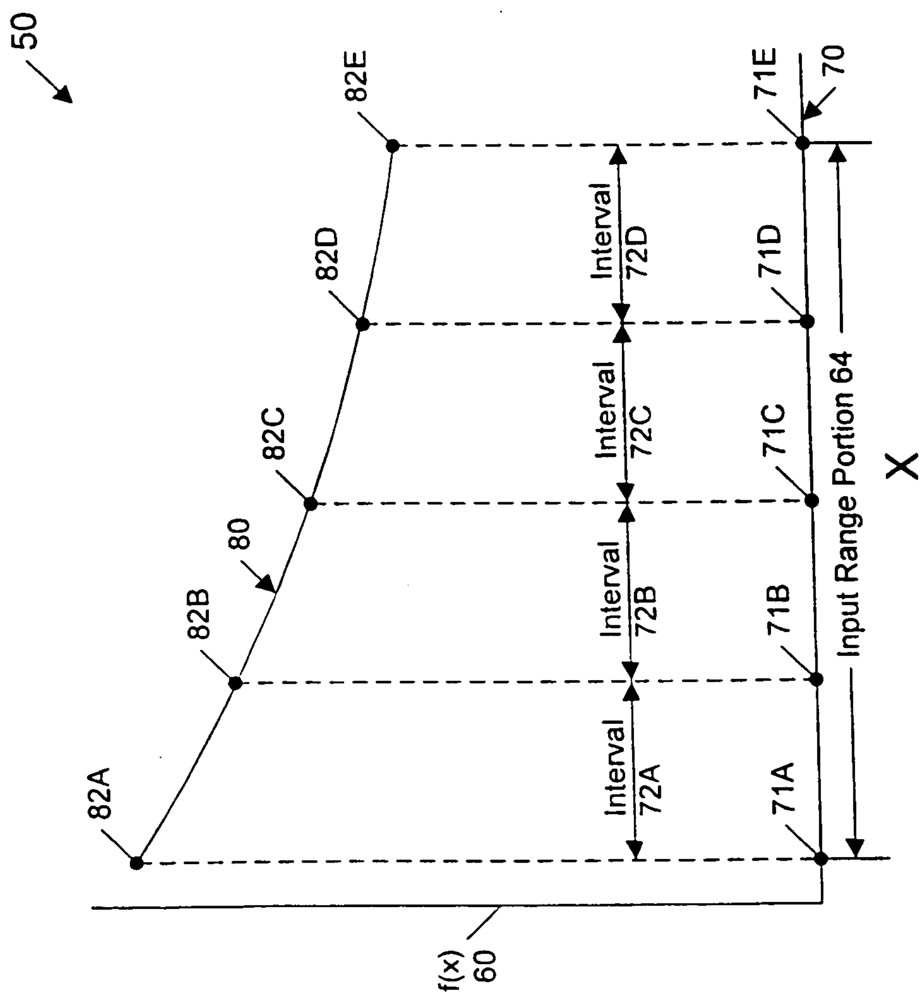


FIG. 53
(PRIOR ART)

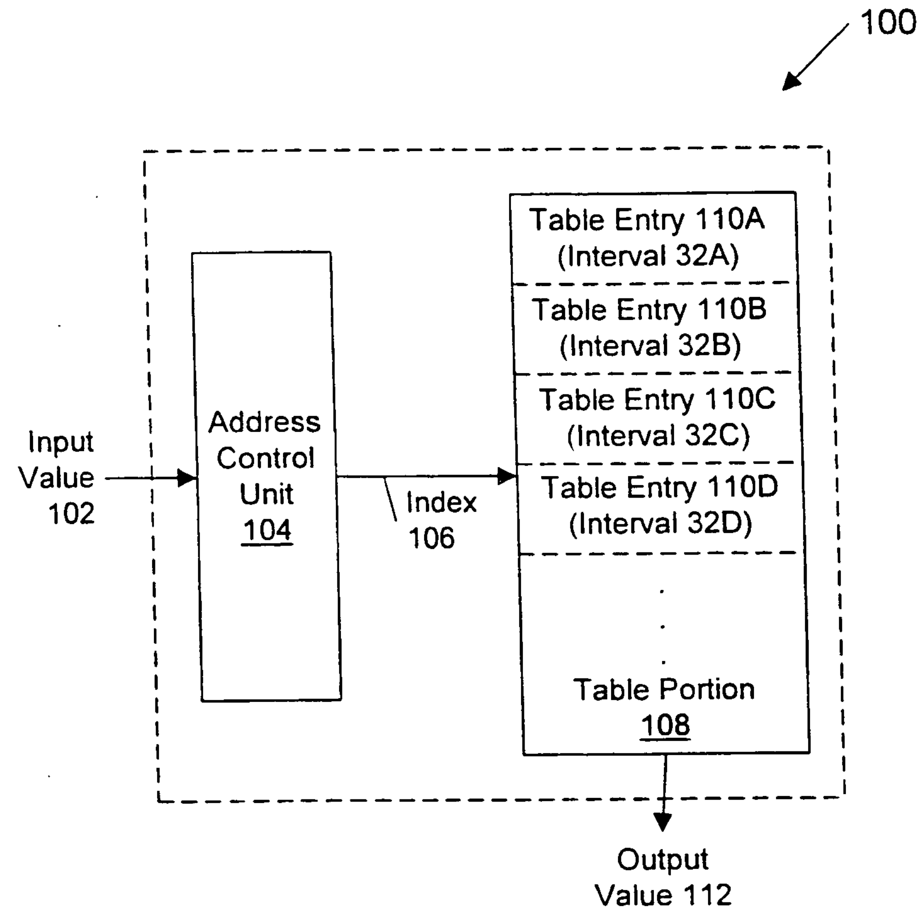


FIG. 54
(PRIOR ART)

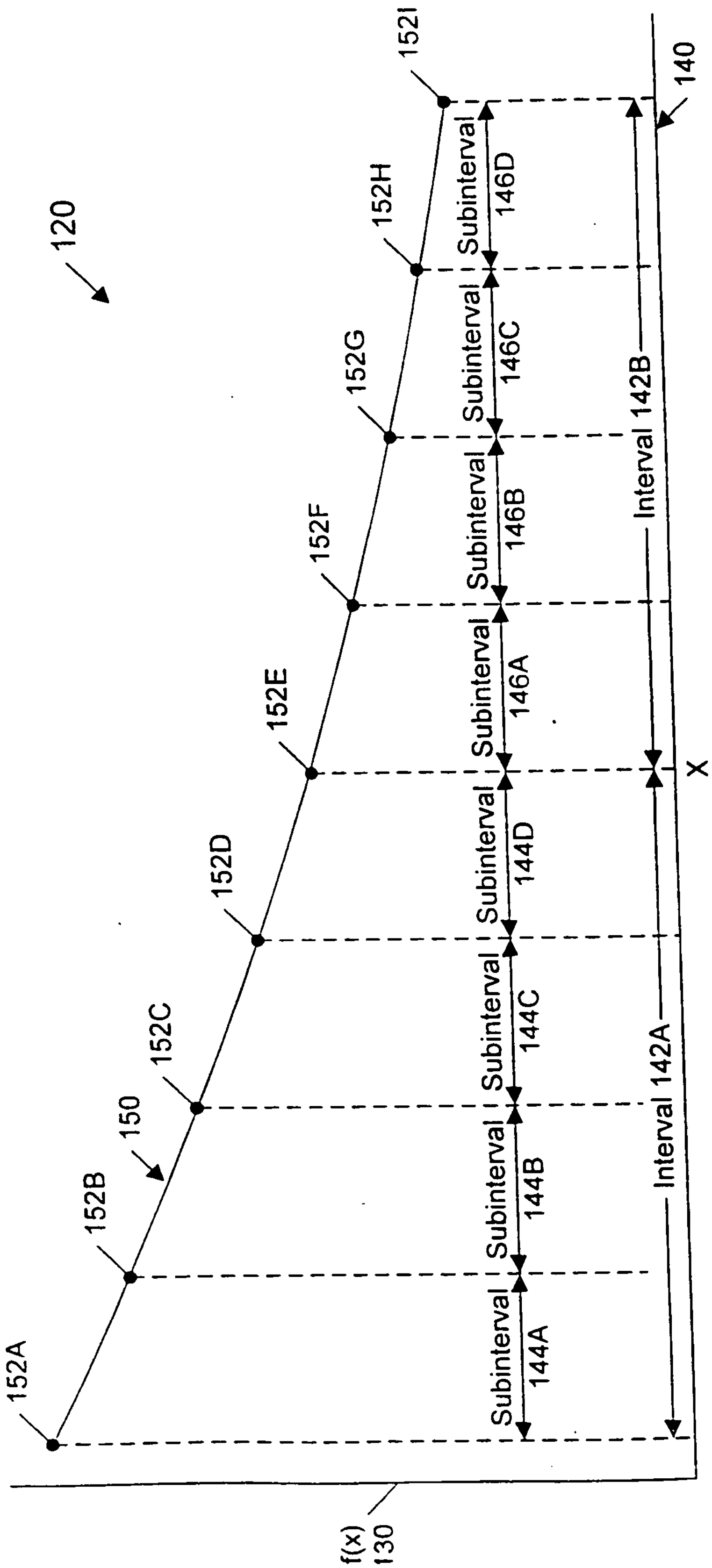


FIG. 55
(PRIOR ART)

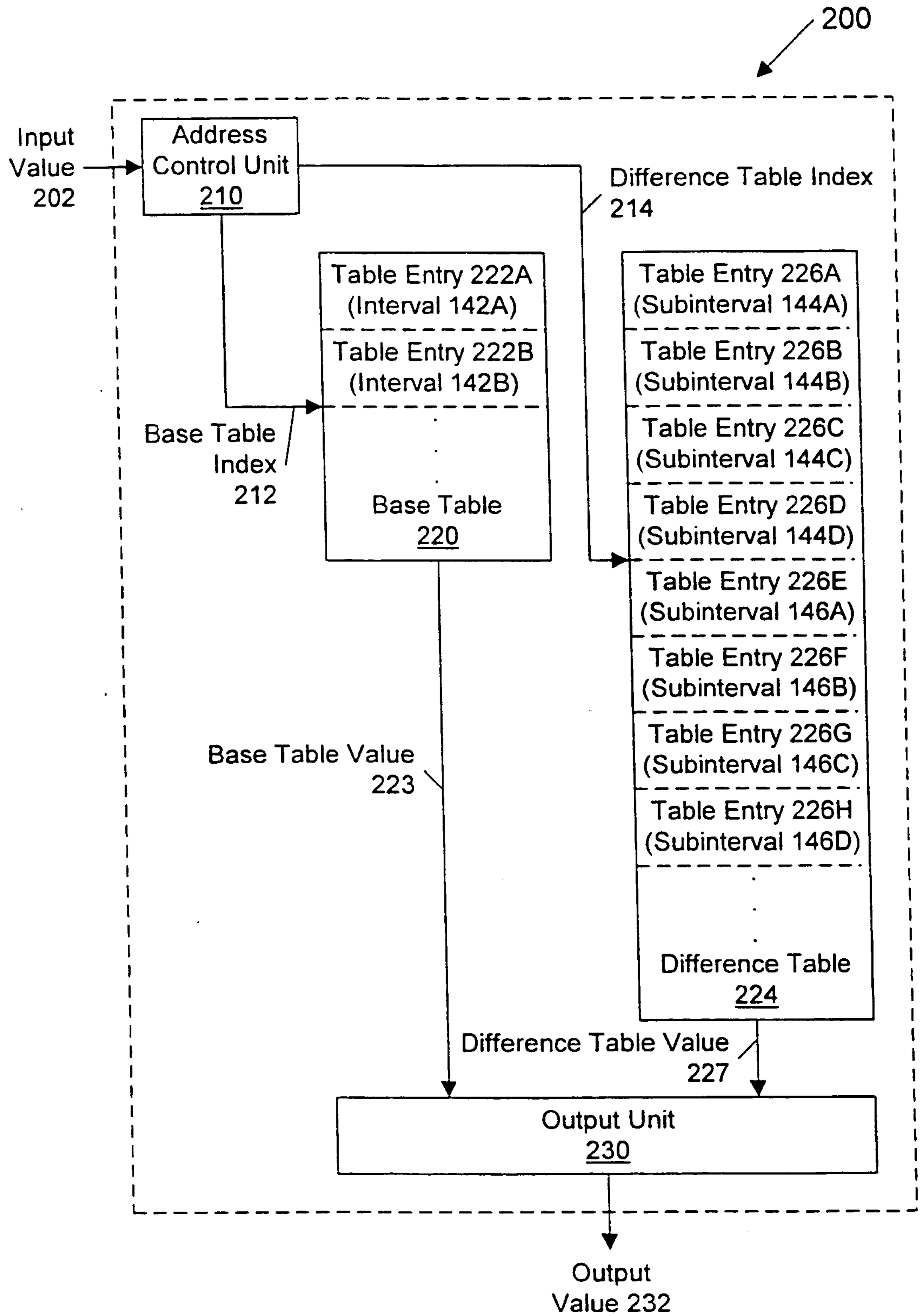


FIG. 56
(PRIOR ART)

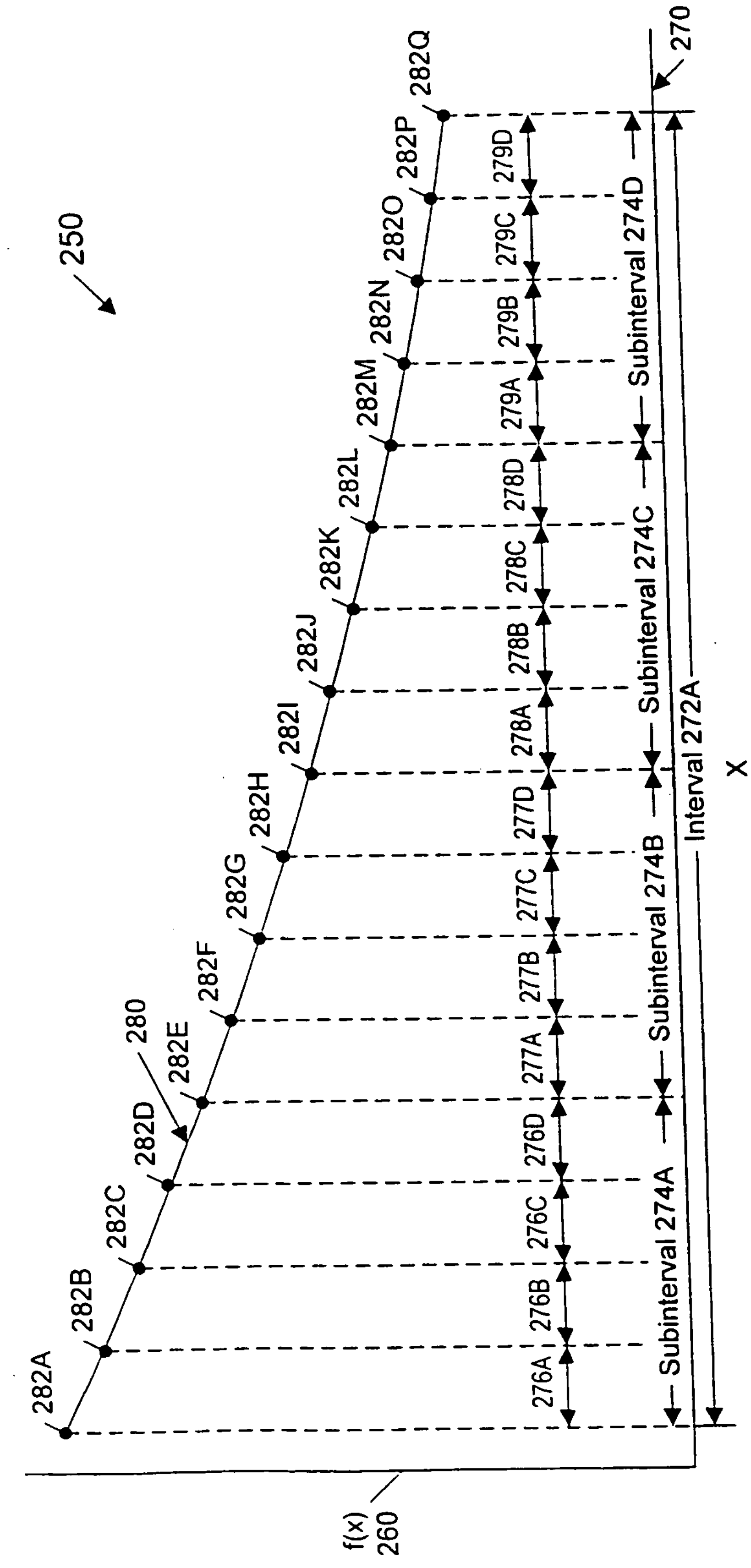

250
270
280
282A
282B
282C
282D
282E
282F
282G
282H
282I
282J
282K
282L
282M
282N
282O
282P
282Q
276A
276B
276C
276D
277A
277B
277C
277D
278A
278B
278C
278D
279A
279B
279C
279D
Subinterval 274A
Subinterval 274B
Subinterval 274C
Subinterval 274D
Interval 272A
X
f(x)
260

FIG. 57

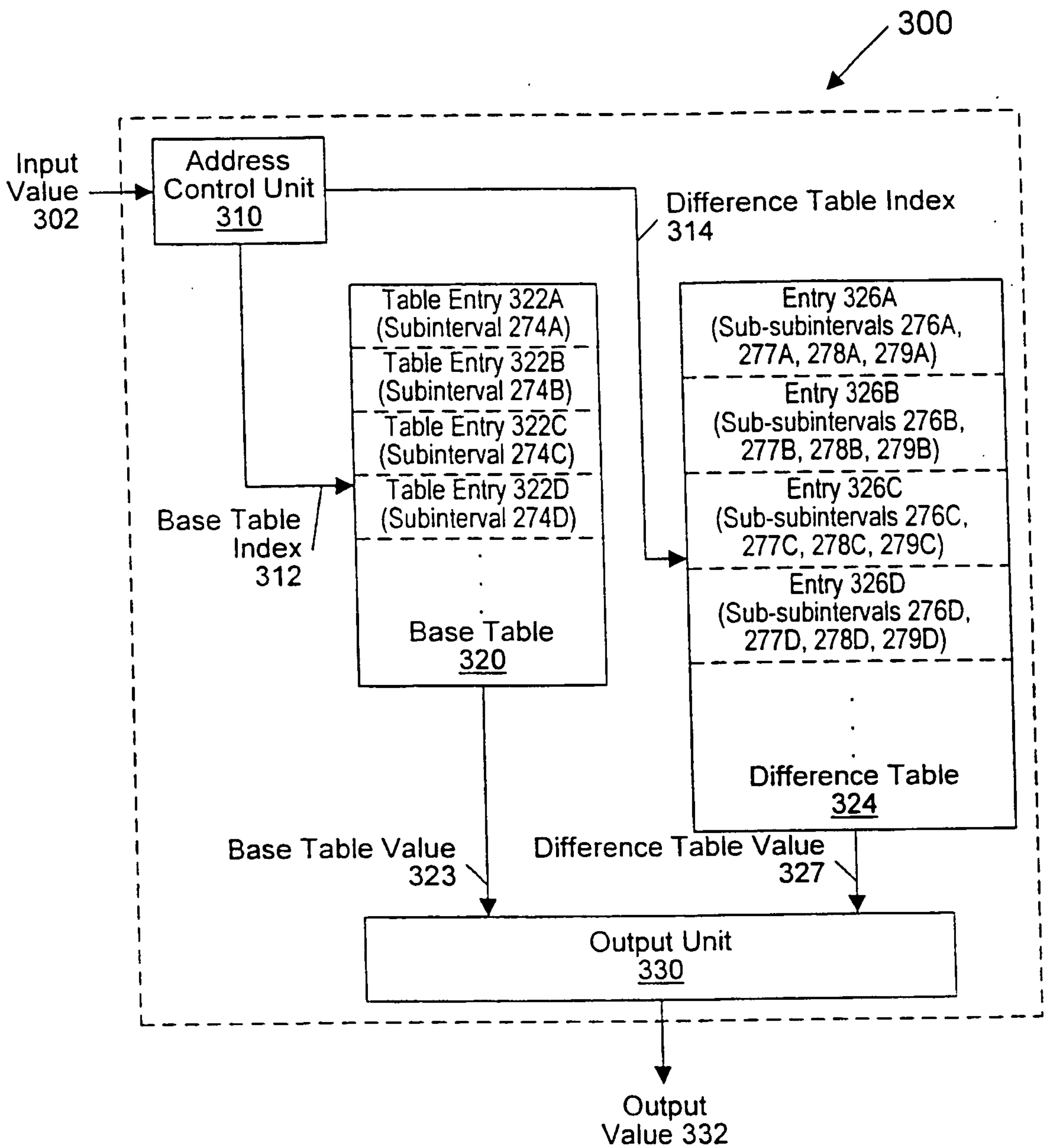

300
Input Value 302
Address Control Unit 310
Difference Table Index 314
Table Entry 322A (Subinterval 274A)
Table Entry 322B (Subinterval 274B)
Table Entry 322C (Subinterval 274C)
Table Entry 322D (Subinterval 274D)
Base Table 320
Base Table Index 312
Entry 326A (Sub-subintervals 276A, 277A, 278A, 279A)
Entry 326B (Sub-subintervals 276B, 277B, 278B, 279B)
Entry 326C (Sub-subintervals 276C, 277C, 278C, 279C)
Entry 326D (Sub-subintervals 276D, 277D, 278D, 279D)
Difference Table 324
Base Table Value 323
Difference Table Value 327
Output Unit 330
Output Value 332

FIG. 58

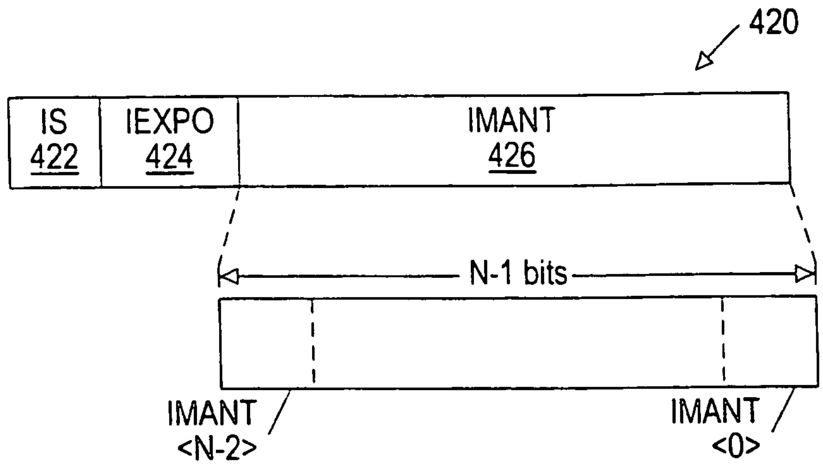


FIG. 60A

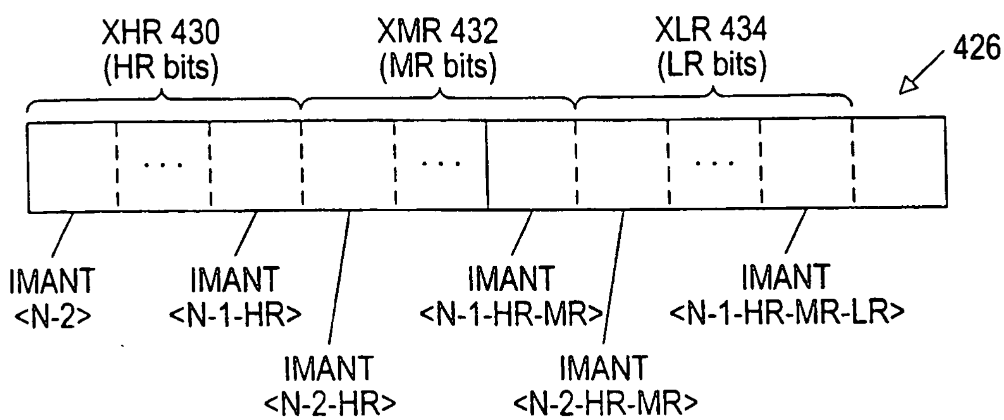


FIG. 60B

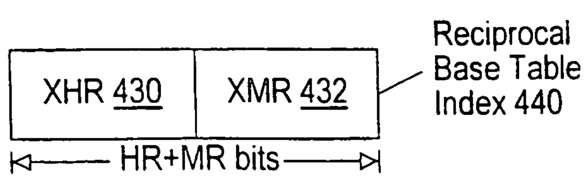


FIG. 60C

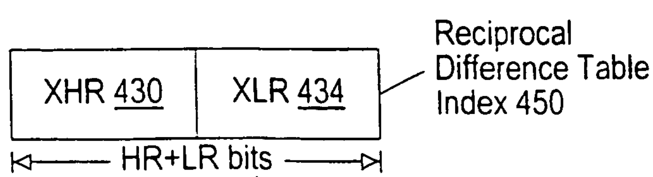


FIG. 60D

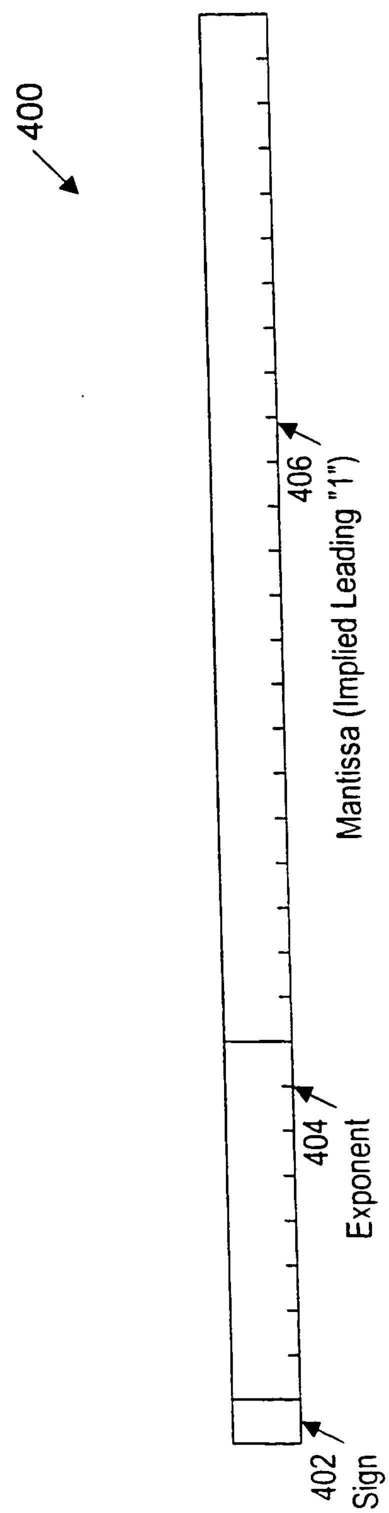


FIG. 59

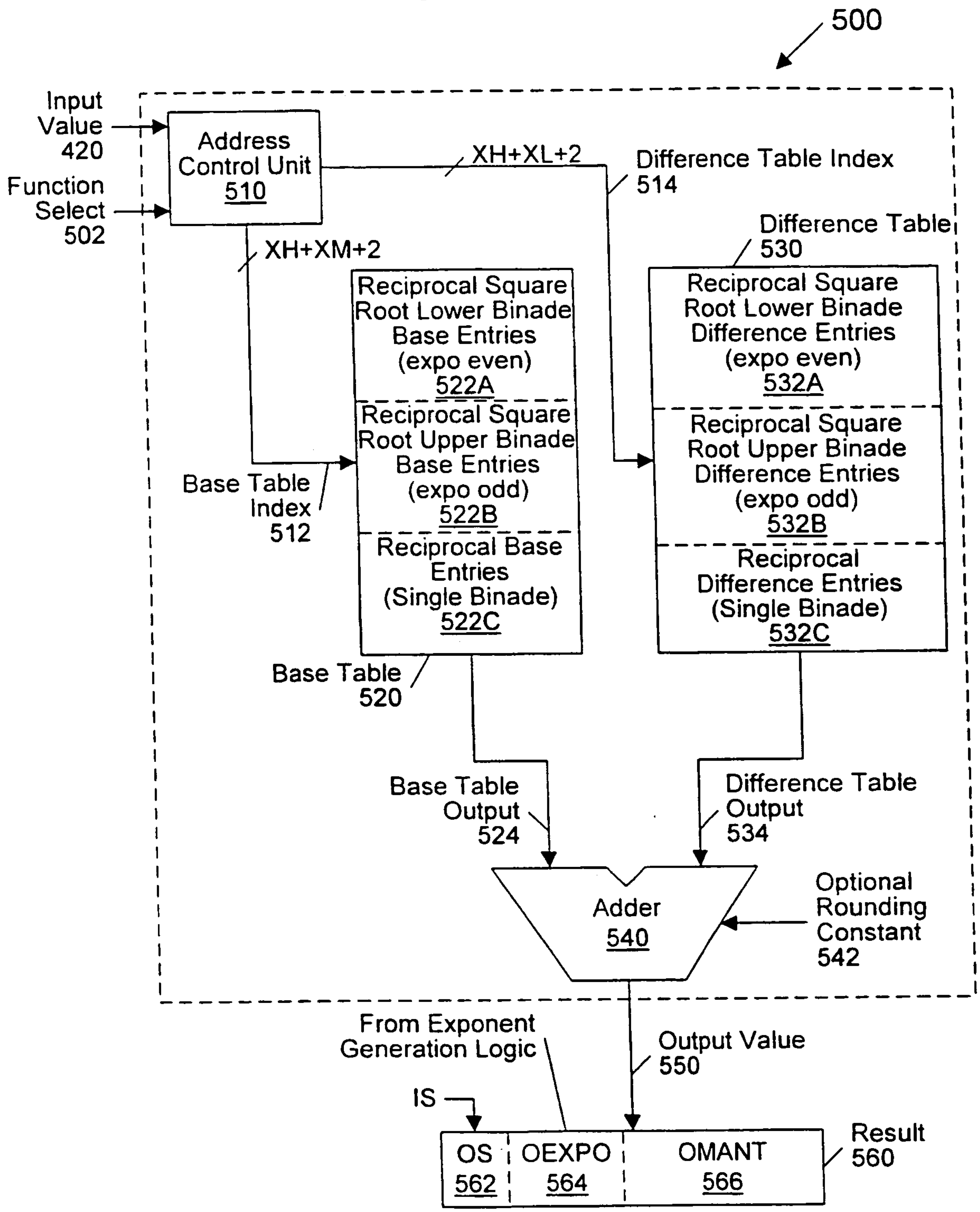

500
Input Value 420
Function Select 502
Address Control Unit 510
XH+XL+2
Difference Table Index 514
XH+XM+2
Difference Table 530
Reciprocal Square Root Lower Binade Base Entries (expo even) 522A
Reciprocal Square Root Upper Binade Base Entries (expo odd) 522B
Reciprocal Base Entries (Single Binade) 522C
Base Table Index 512
Reciprocal Square Root Lower Binade Difference Entries (expo even) 532A
Reciprocal Square Root Upper Binade Difference Entries (expo odd) 532B
Reciprocal Difference Entries (Single Binade) 532C
Base Table 520
Base Table Output 524
Difference Table Output 534
Adder 540
Optional Rounding Constant 542
From Exponent Generation Logic
Output Value 550
IS
OS 562
OEXPO 564
OMANT 566
Result 560

FIG. 62

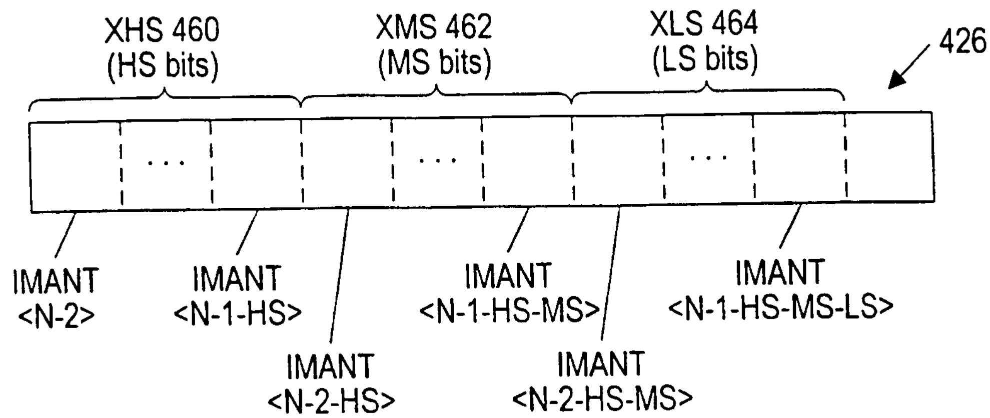


FIG. 61A

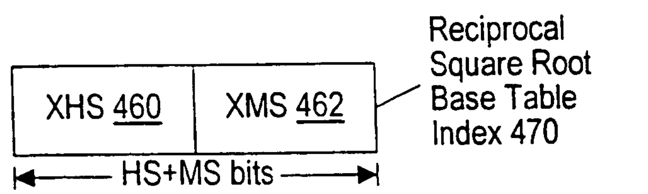


FIG. 61B

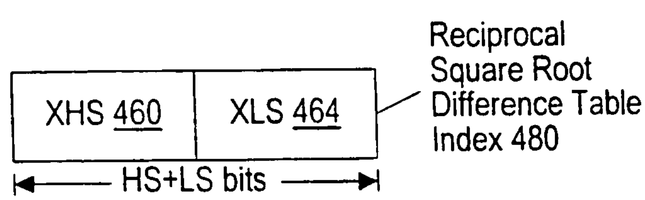


FIG. 61C

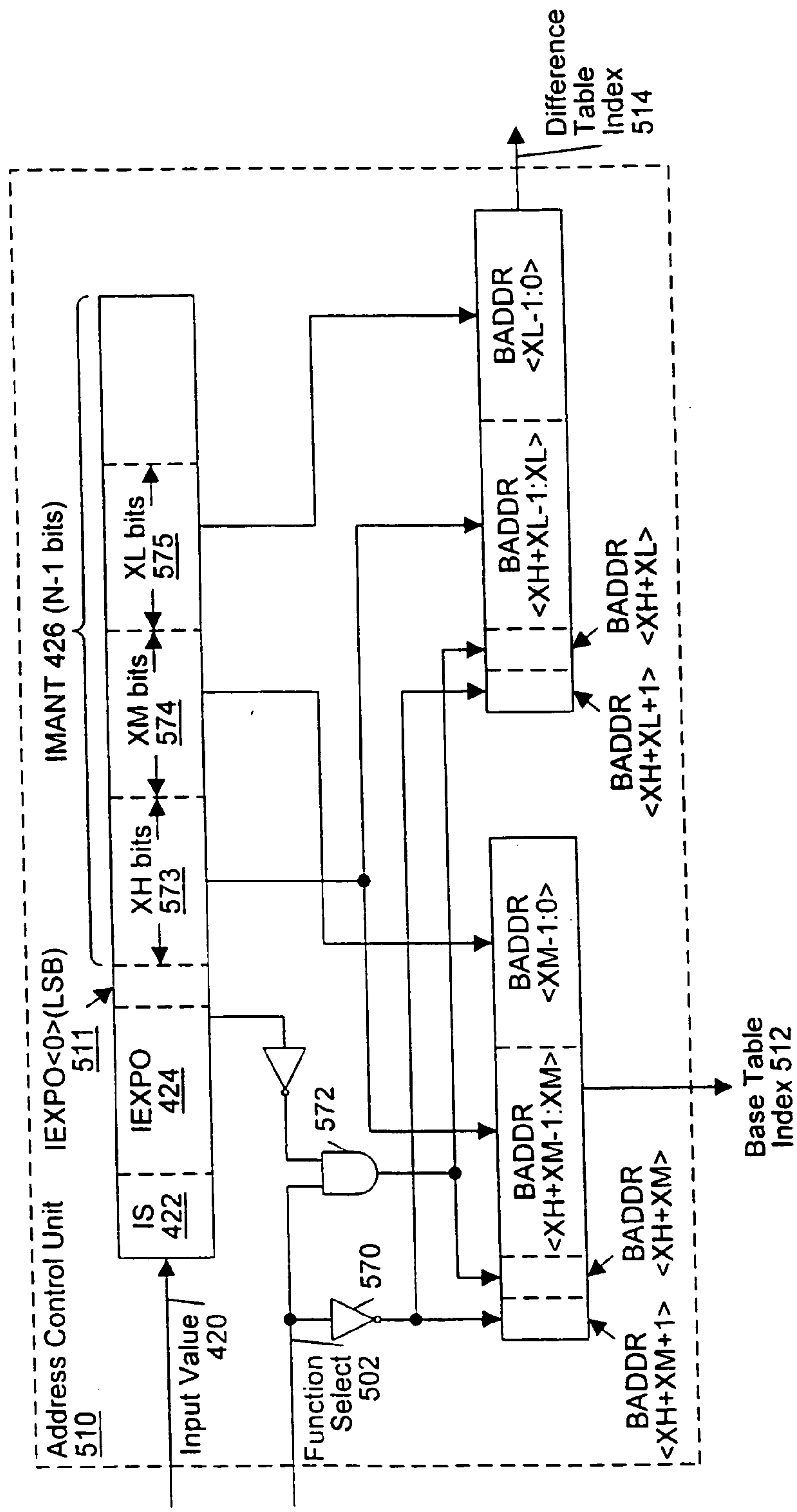

Difference Table Index 514
IMANT 426 (N-1 bits)
XL bits 575
XM bits 574
XH bits 573
IEXPO<0>(LSB) 511
IEXPO 424
IS 422
Address Control Unit 510
Input Value 420
Function Select 502
570
572
BADDR <XL-1:0>
BADDR <XH+XL-1:XL>
BADDR <XH+XL>
BADDR <XH+XL+1>
BADDR <XM-1:0>
BADDR <XH+XM-1:XM>
BADDR <XH+XM>
BADDR <XH+XM+1>
Base Table Index 512

FIG. 63

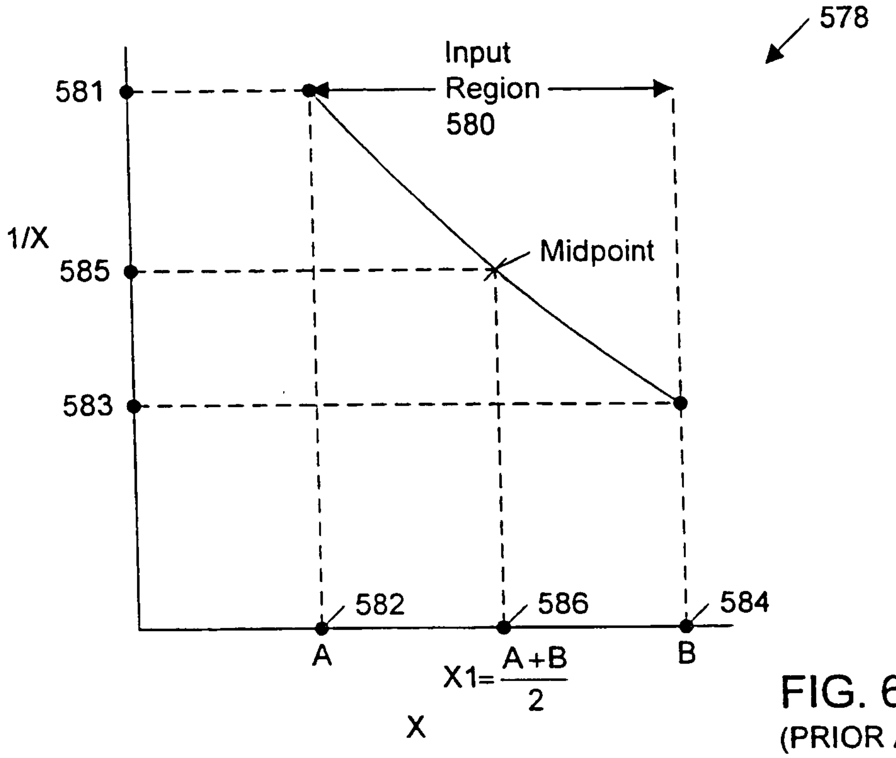


FIG. 64A
(PRIOR ART)

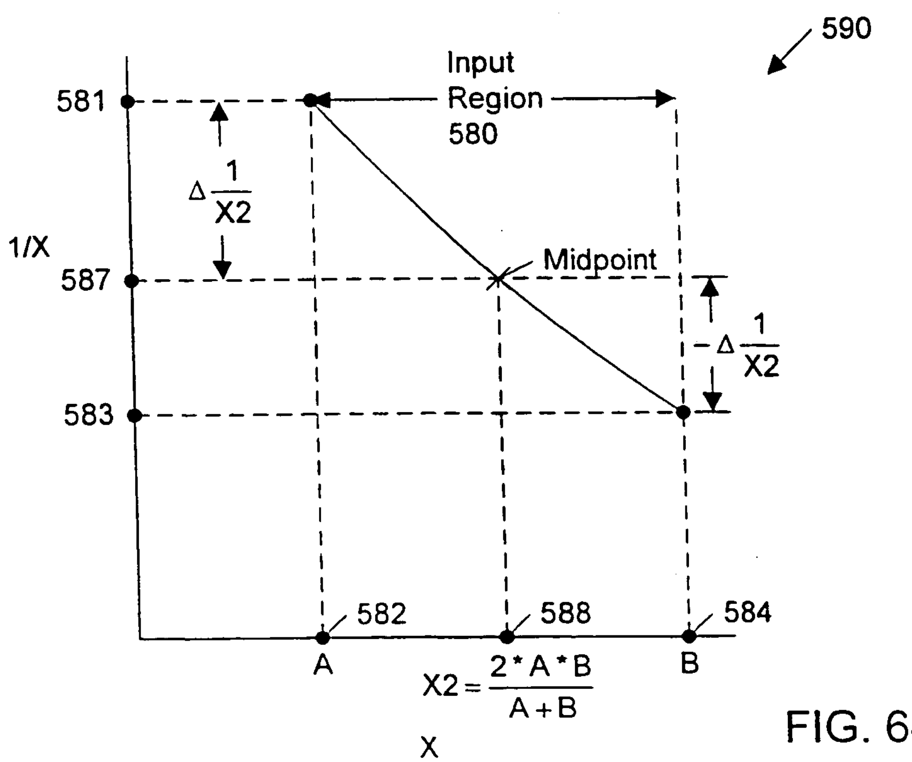


FIG. 64B

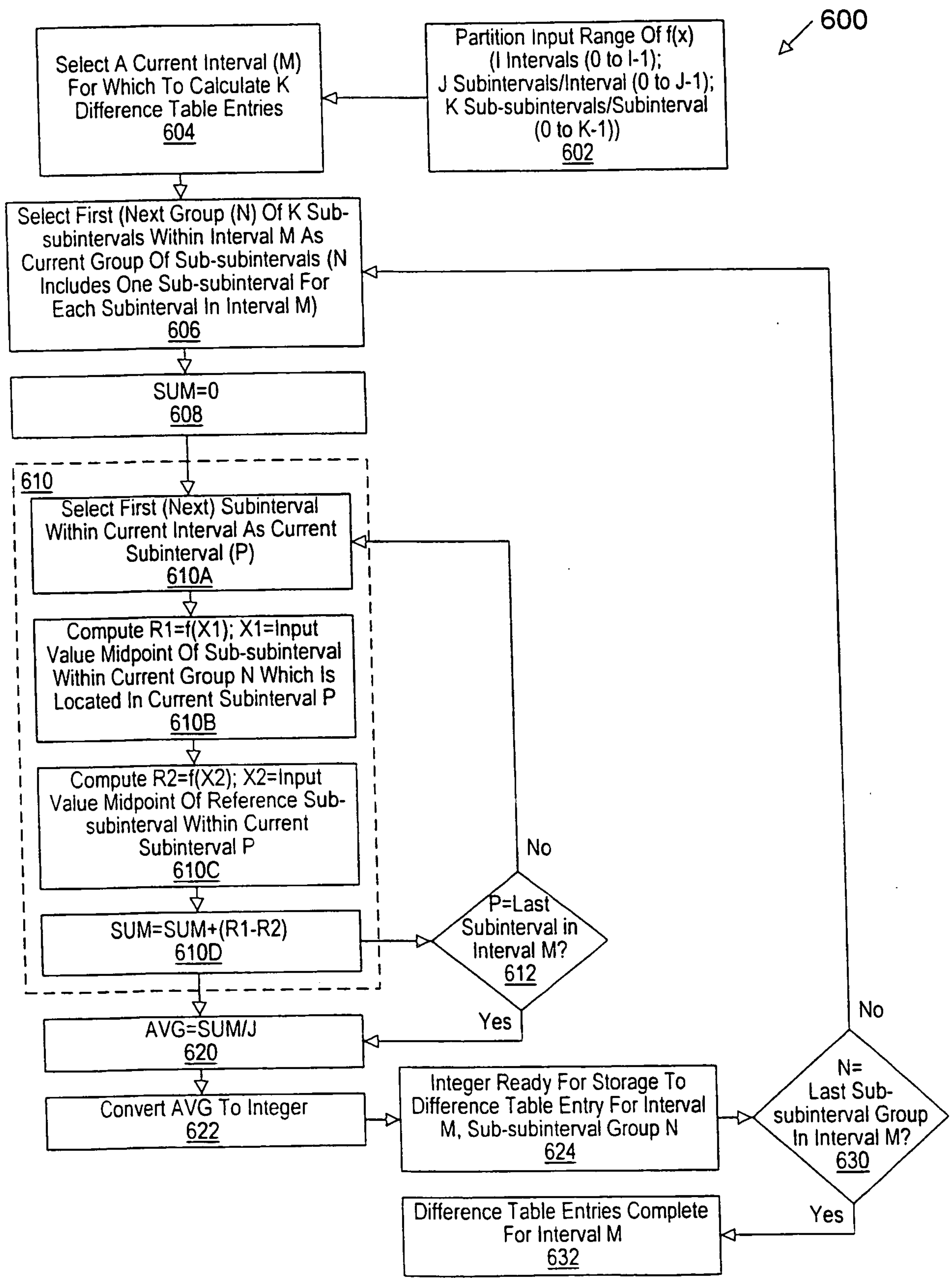

600
Partition Input Range Of f(x) (I Intervals (0 to I-1); J Subintervals/Interval (0 to J-1); K Sub-subintervals/Subinterval (0 to K-1)) 602
Select A Current Interval (M) For Which To Calculate K Difference Table Entries 604
Select First (Next Group (N) Of K Sub-subintervals Within Interval M As Current Group Of Sub-subintervals (N Includes One Sub-subinterval For Each Subinterval In Interval M) 606
SUM=0 608
610
Select First (Next) Subinterval Within Current Interval As Current Subinterval (P) 610A
Compute R1=f(X1); X1=Input Value Midpoint Of Sub-subinterval Within Current Group N Which Is Located In Current Subinterval P 610B
Compute R2=f(X2); X2=Input Value Midpoint Of Reference Sub-subinterval Within Current Subinterval P 610C
SUM=SUM+(R1-R2) 610D
P=Last Subinterval in Interval M? 612
No
Yes
AVG=SUM/J 620
Convert AVG To Integer 622
Integer Ready For Storage To Difference Table Entry For Interval M, Sub-subinterval Group N 624
N= Last Sub-subinterval Group In Interval M? 630
No
Yes
Difference Table Entries Complete For Interval M 632

FIG. 65A

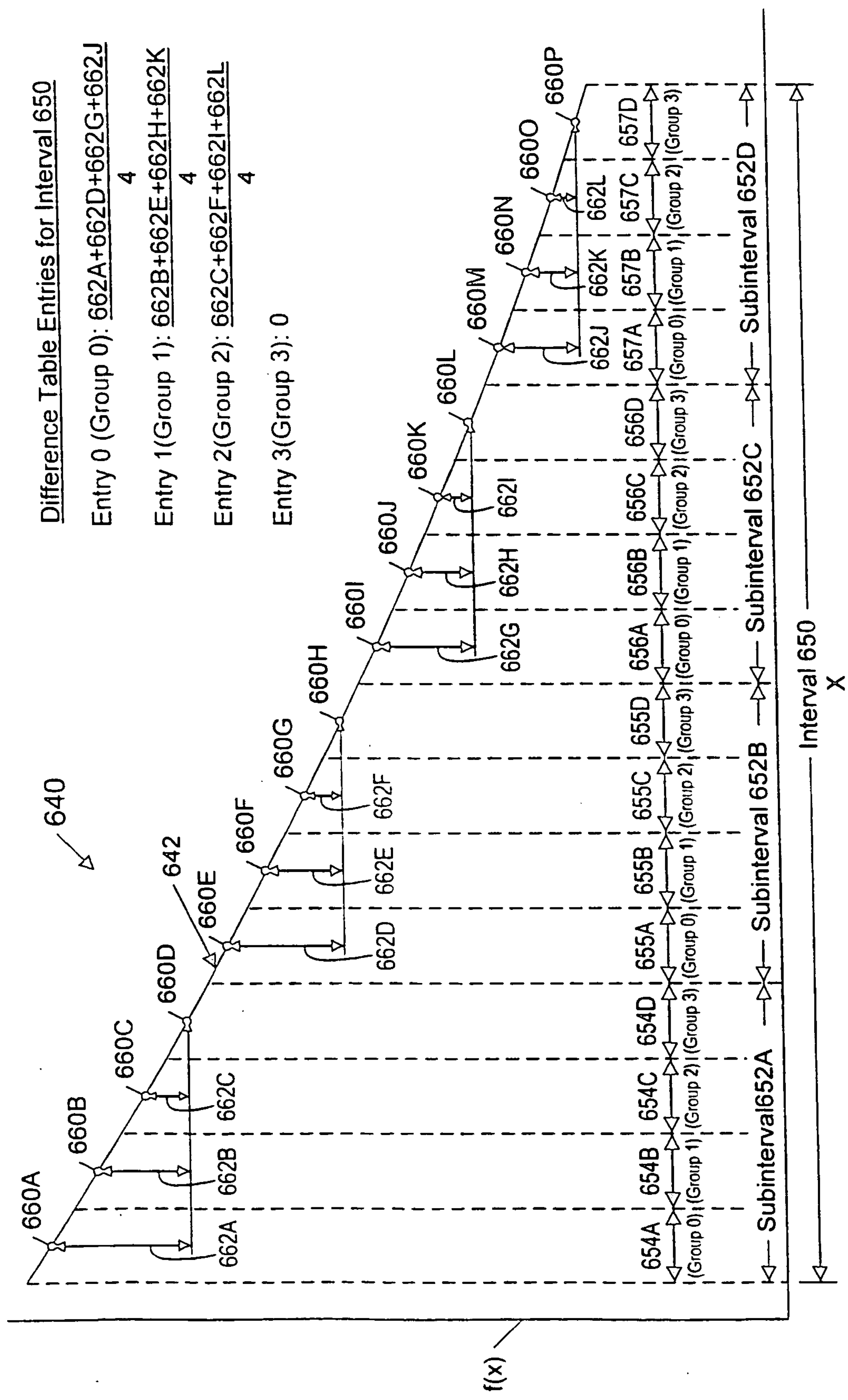

Difference Table Entries for Interval 650
Entry 0 (Group 0): 662A+662D+662G+662J / 4
Entry 1(Group 1): 662B+662E+662H+662K / 4
Entry 2(Group 2): 662C+662F+662I+662L / 4
Entry 3(Group 3): 0
640
642
660A
660B
660C
660D
660E
660F
660G
660H
660I
660J
660K
660L
660M
660N
660O
660P
662A
662B
662C
662D
662E
662F
662G
662H
662I
662J
662K
662L
654A
654B
654C
654D
655A
655B
655C
655D
656A
656B
656C
656D
657A
657B
657C
657D
(Group 0)
(Group 1)
(Group 2)
(Group 3)
Subinterval652A
Subinterval 652B
Subinterval 652C
Subinterval 652D
Interval 650
X
f(x)

FIG. 65B

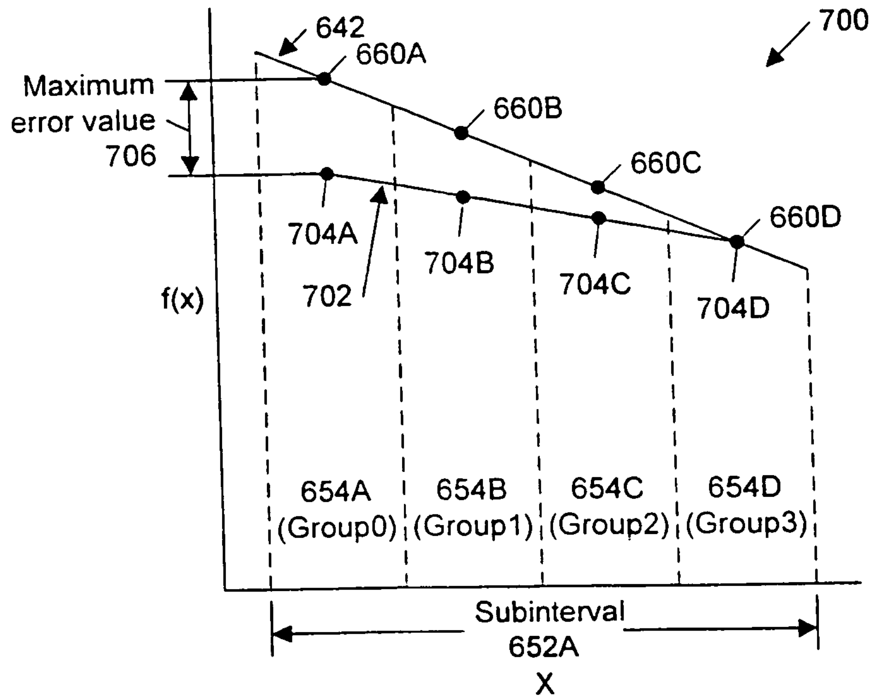


FIG. 66A

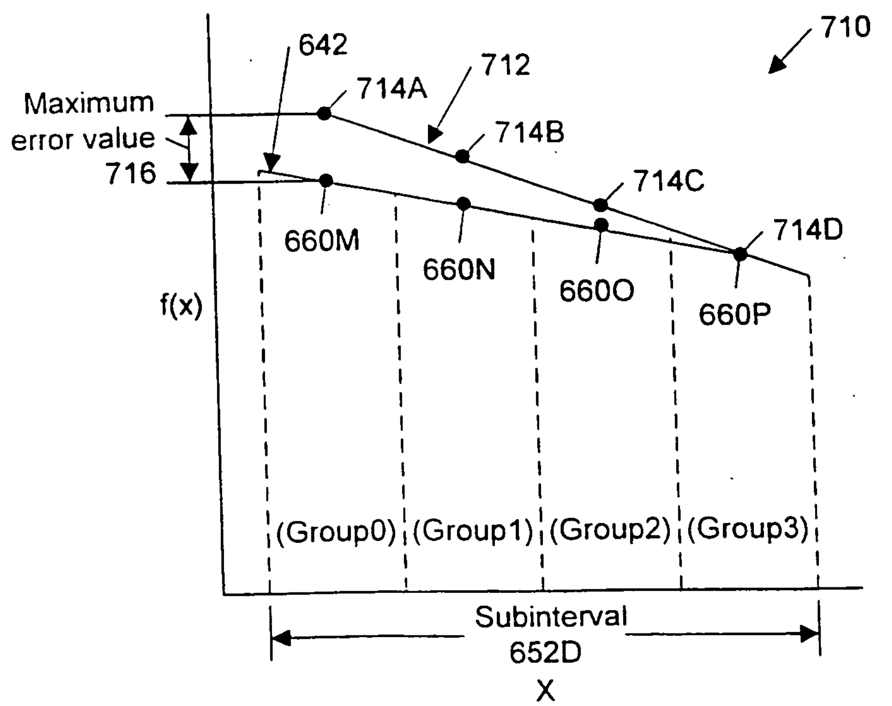


FIG. 66B

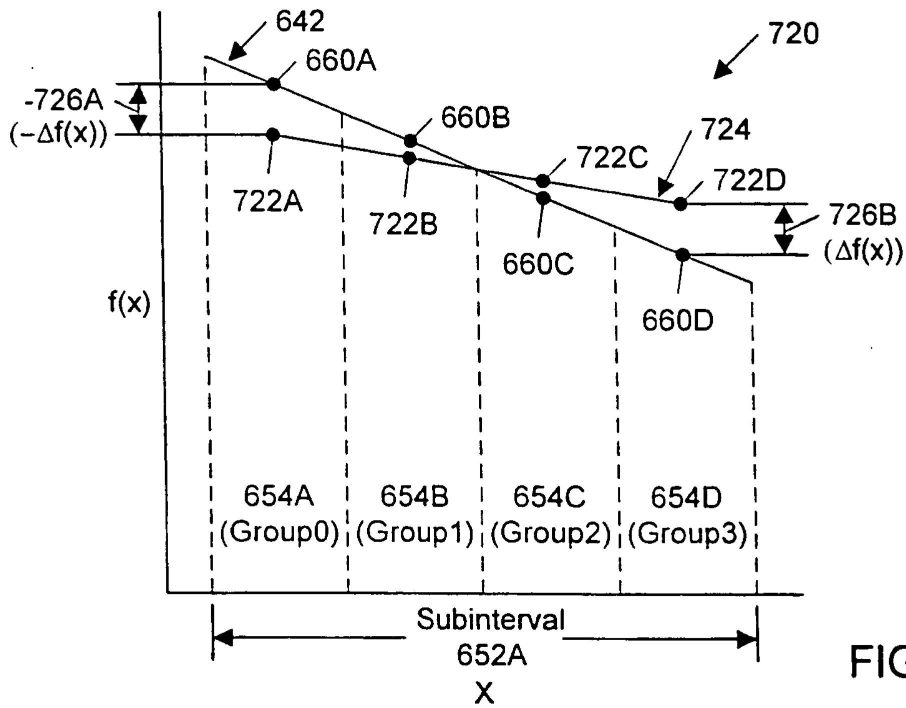


FIG. 66C

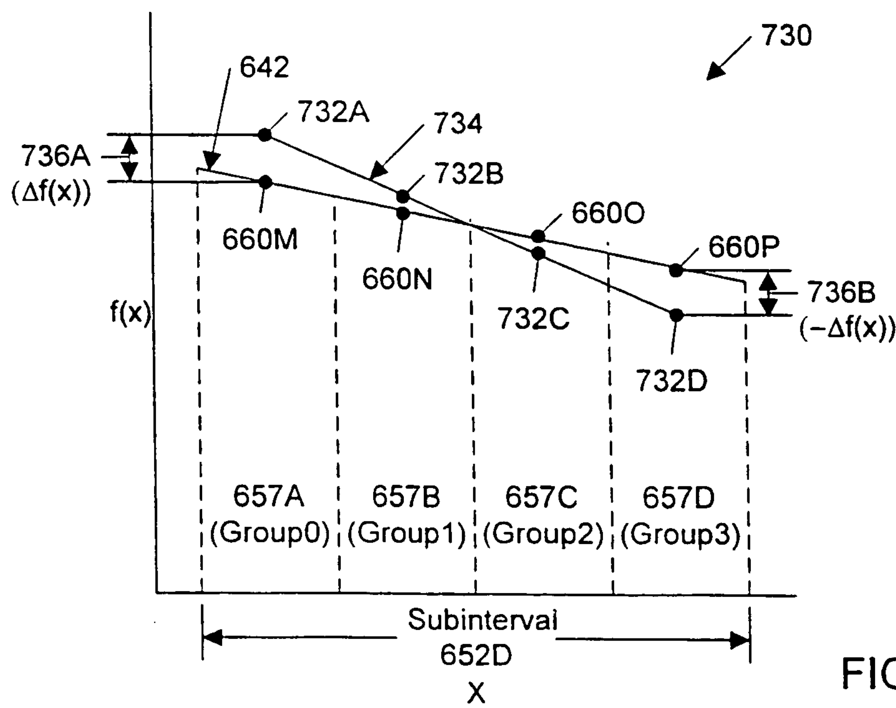


FIG. 66D

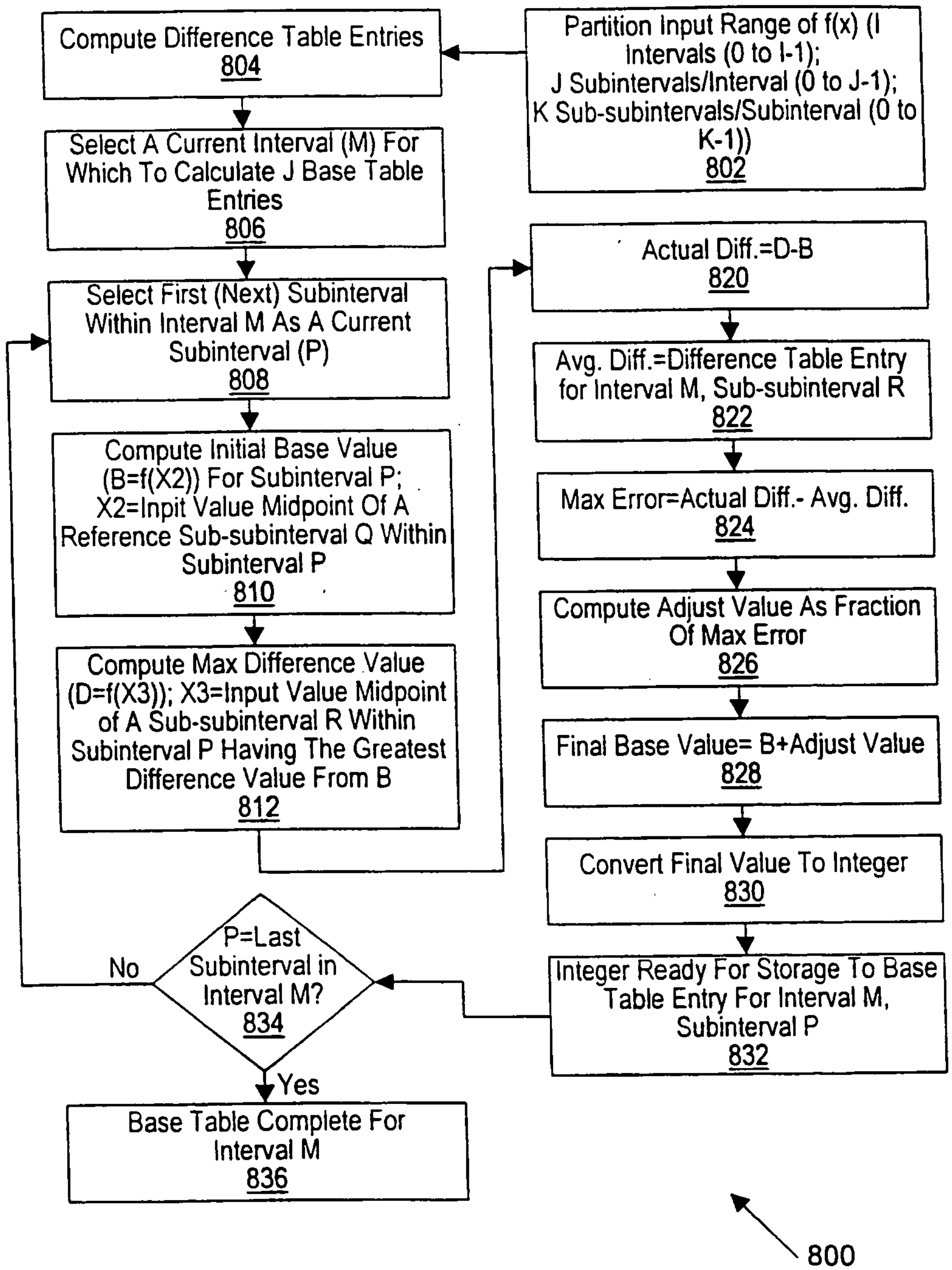

Partition Input Range of f(x) (I Intervals (0 to I-1); J Subintervals/Interval (0 to J-1); K Sub-subintervals/Subinterval (0 to K-1)) 802
Compute Difference Table Entries 804
Select A Current Interval (M) For Which To Calculate J Base Table Entries 806
Select First (Next) Subinterval Within Interval M As A Current Subinterval (P) 808
Compute Initial Base Value (B=f(X2)) For Subinterval P; X2=Inpit Value Midpoint Of A Reference Sub-subinterval Q Within Subinterval P 810
Compute Max Difference Value (D=f(X3)); X3=Input Value Midpoint of A Sub-subinterval R Within Subinterval P Having The Greatest Difference Value From B 812
Actual Diff.=D-B 820
Avg. Diff.=Difference Table Entry for Interval M, Sub-subinterval R 822
Max Error=Actual Diff.- Avg. Diff. 824
Compute Adjust Value As Fraction Of Max Error 826
Final Base Value= B+Adjust Value 828
Convert Final Value To Integer 830
Integer Ready For Storage To Base Table Entry For Interval M, Subinterval P 832
P=Last Subinterval in Interval M? 834
No
Yes
Base Table Complete For Interval M 836
800

FIG. 67

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 1/035, 7/50, 7/00, H03M 7/24
// G06F 101:08, 101:12

A3

(11) International Publication Number: WO 99/23548

(43) International Publication Date: 14 May 1999 (14.05.99)

(21) International Application Number: PCT/US98/22453

(22) International Filing Date: 22 October 1998 (22.10.98)

(30) Priority Data:

60/063,601	23 October 1997 (23.10.97)	US
60/063,600	23 October 1997 (23.10.97)	US
09/015,084	29 January 1998 (29.01.98)	US
09/049,851	27 March 1998 (27.03.98)	US
09/049,863	27 March 1998 (27.03.98)	US
09/049,893	27 March 1998 (27.03.98)	US
09/049,750	27 March 1998 (27.03.98)	US
09/049,758	27 March 1998 (27.03.98)	US
09/055,916	6 April 1998 (06.04.98)	US
09/098,482	16 June 1998 (16.06.98)	US

(71) Applicant: ADVANCED MICRO DEVICES, INC. [US/US]; One AMD Place, Mail Stop 68, Sunnyvale, CA 94088–3453 (US).

(72) Inventors: OBERMAN, Stuart, F.; 265 G. South Bay View Avenue #G, Sunnyvale, CA 94086 (US). JUFFA, Norbert; 460 Navaro Way #201, San Jose, CA 95134 (US). ROBERTS, Mark, E.; 2626 Ohlone Drive, San Jose, CA 95132 (US). WEBER, Frederick, D.; 1137 Settle Avenue, San Jose, CA 95125 (US). CHERUKURI, Ravi, Krishna; 952 Big Bear Court, Milpitas, CA 95035 (US). RAMANI, Krishnan; 655 South Fair Oaks Avenue #G201, Sunnyvale, CA 94086 (US).

(74) Agent: KIVLIN, B., Noël; Conley, Rose & Tayon, P.C., P.O. Box 398, Austin, TX 78767–0398 (US).

(81) Designated States: CN, JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(88) Date of publication of the international search report: 5 August 1999 (05.08.99)

(54) Title: MULTIFUNCTION FLOATING POINT ADDITION/SUBTRACTION PIPELINE AND BIPARTITE LOOK-UP TABLE

(57) Abstract

An add/subtract pipeline has far and close data paths. The far data path handles effective addition operations, and effective subtraction operations for operands having an absolute exponent difference greater than one. The close data path handles all other effective subtraction operations. Selection of the output value in the close data path effectuates the round-to-nearest operation. Floating point-to-integer conversion may be executed in the far data path integer-to-floating point instructions in the close data path. The execution unit may include a pluraliy of add/subtract pipelines, allowing vectored add, subtract, and integer/floating point conversion instructions to be performed. Additional arithmetic instructions (such as reverse subtract and accumulate functions minimum/maximum and comparison) may also be implemented. A method for generating entries for a bipartite look-up table having base and difference table portions is also disclosed. So is a multi-function look-up table.

INTERNATIONAL SEARCH REPORT

rnational Application No
PCT/US 98/22453

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F1/035 G06F7/50 G06F7/00 H03M7/24
//G06F101:08,G06F101:12

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	D. DAS SARMA ET AL.: "FAITHFUL BIPARTITE ROM RECIPROCAL TABLES" PROCEEDINGS OF THE 12TH SYMPOSIUM ON COMPUTER ARITHMETIC, BATH, 19 - 21 JULY 1995, no. SYMP. 12, 19 July 1995, pages 17-28, XP000548629 IEEE COMPUTER SOCIETY, LOS ALAMITOS, CA. USA see the whole document --- -/--	1-3,6,7

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 January 1999	04. 06. 1999

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	VERHOOF, P

3

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 98/22453

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	H. HASSLER ET AL.: "FUNCTION EVALUATION BY TABLE LOOK-UP AND ADDITION" PROCEEDINGS OF THE 12TH SYMPOSIUM ON COMPUTER ARITHMETIC, BATH, 19 - 21 JULY 1995, no. SYMP. 12, 19 July 1995, pages 10-16, XP000548628 IEEE COMPUTER SOCIETY, LOS ALAMITOS, CA. USA see page 13 ---	1-9
A	M. SCHULTE ET AL.: "Symmetric Bipartite Tables for Accurate Function Approximation" PROCEEDINGS OF THE 13TH SYMPOSIUM ON COMPUTER ARITHMETIC, ASILOMAR, CA., USA, 6-9 JULY 1997,6 July 1997, pages 175-183, XP002091465 IEEE COMPUTER SOCIETY, LOS ALAMITOS, CA. USA see the whole document ---	1-9
A	WO 94 18632 A (PICKETT LESTER CARYL) 18 August 1994 see figure 1 ---	1-9
A	US 5 369 607 A (OKAMOTO FUYUKI) 29 November 1994 see column 7, line 48 - column 8, line 45; figure 6 -----	1

3